A Semantic Least Privilege and Semi-Automated Approach to Preventing Cyber Attacks on Web Applications

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Stuart Steiner

Major Professor: **Dr. Daniel Conte de Leon**

Committee Members: **Dr. Jim Alves-Foss**, **Dr. Robert Rinker**, **Dr. Yacine Chakhchoukh**

Department Administrator: **Dr. Terence Soule**

August 2018

Authorization to Submit Dissertation

This dissertation of **Stuart Steiner**, submitted for the degree of Doctor of Philosophy with a major in Computer Science and entitled **"A Semantic Least Privilege and Semi-Automated Approach to Preventing Cyber Attacks on Web Applications,"** has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: ________________________________ Date: ____________
Dr. Daniel Conte de Leon

Committee Member: ________________________________ Date: ____________
Dr. Jim Alves-Foss

Committee Member: ________________________________ Date: ____________
Dr. Robert Rinker

Committee Member: ________________________________ Date: ____________
Dr. Yacine Chakhchoukh

Department Administrator: ________________________________ Date: ____________
Dr. Terence Soule

Abstract

Structured Query Language injection attacks still remain one of the most commonly occurring and exploited types of web application vulnerabilities. A considerable amount of research concerning Structured Query Language injection attacks mitigation techniques has found that the primary solution requires developers to utilize secure development techniques. However, the standard practice for many current web applications, including web application coding tutorials, does not implement well-known secure design principles or secure development techniques.

Because most websites do not use secure development techniques or do not apply them correctly, within the last three years, hundreds of millions of private data records have been compromised in high-profile data breaches, resulting in billions of dollars in economic losses and unrecoverable privacy losses. One commonality of the data breaches is the standard practice, in a web application, for the front-end and middleware processes to have root privileges to the complete database management system. This practice is in stark opposition to the well-known secure design principle of least privilege introduced 40 years ago. Enforcing least privilege at all levels of a web application would help prevent and mitigate future data breaches.

This dissertation describes a systematic, semi-automated, formal and repeatable process for converting a web application and its corresponding back-end database from a non-least privilege implementation into a least privilege implementation. The steps needed for this redesign and semi-automated refactoring process are explained through the use of two case studies. Case study one is based on the SEED Labs Structured Query Language injection attack web application. Case study two is based on the OWASP Mutillidae II web application. Each case study also describes the formal access control modeling and associated toolset used to aid and partially automate this systematic conversion.

The evaluation of the results suggests that this novel process is effective at modeling web applications security policies, as well as mitigating and preventing attacks. With the help of the modeling and automation capabilities provided by this approach and associated toolset, least-privilege-based web application hardening may be implemented by web developers on current and new web applications regardless of their knowledge of secure design principles. This novel systematic modeling approach shows great promise toward helping web developers better understand the security model of web applications. Furthermore, the associated toolset may lead to further automating the web application hardening process through the application of the principle of least privilege.

Acknowledgments

I would like to extend my gratitude to my major professor, Dr. Daniel Conte de Leon. His guidance concerning this dissertation has been invaluable to me time and time again. His patience and direction have always kept me pushing forward.

In addition, I would like to thank my committee members, Dr. Jim Alves-Foss, Dr. Robert Rinker, and Dr. Yacine Chakhchoukh for their acceptance, guidance, encouragement and input in the process of completing this dissertation.

I would also like to thank Ananth Jillepalli for being an incredible research partner that has always been there to discuss ideas, encourage me and for being great person.

Finally, I would like to thank Arvilla Daffin for being there to provide the proper guidance. I would also like to thank every student past and present that I worked with in some capacity. It was always a pleasure in whatever we were working on.

Dedication

I would like to thank my beautiful wife Shirlee Steiner. She has always been by my side encouraging me and pushing me. The last 10 years have not been an easy journey and I could not have completed the journey without your continuous and unconditional love and support. Without you this journey would not have been remotely possible.

Table of Contents

List of Figures

List of Tables

List of Code Listings

List of Acronyms

DAG Directed Acyclic Graph

DB Database

DBMS Database Management Systems

DSL Domain Specific Language

FQN Fully Qualified Name

FSA Finite State Automata

HERMES High-Level Easily Reconfigurable Machine Environment Specification

HPol The Hierarchical Policy Model

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IT Information Technology

JSON JavaScript Object Notation

LP HERMES Least Privilege High-Level Easily Reconfigurable Machine Environment Specification

LP PKB Least Privilege Prolog Knowledge Base

Mutillidae The Mutillidae II Web Application

OT Operational Technology

OWASP Open Web Application Security Project

PC Personal Computer

PHP PHP: Hypertext Preprocessor

PID Policy Identifier

POLP The Principle of Least Privilege

SQL Structured Query Language

SQLIA Structured Query Language Injection Attack

SQLIAD Structured Query Language Injection Attack Detection

SQLIAP Structured Query Language Injection Attack Prevention

URL Uniform Resource Locator

XSB Logic Programming and Deductive Database System

Chapter 1: Introduction

This chapter introduces some of the key challenges in securing existing web applications from the common problem of Structured Query Language Structured Query Language (SQL) Injection [2]. Section 1.1 introduces the key concepts concerning web security and SQL injection attacks. Section 1.2 introduces the proposed solution to preventing cyber attacks on web applications. Section 1.3 discusses the objectives of this dissertation. Section 1.4 discusses the contributions of this dissertation. Section 1.6 provides the structure for the rest of the dissertation.

1.1 The Problem: SQL Injection Attacks in Web Applications

Over the past 18 years, the number of websites has grown from 29 million in 2001 to more than 1.9 billion in 2018 [3]. At the beginning of the 21st century the Web, known as Web 1.0, consisted of websites that were static pages. Around 2002, Web 2.0 was created and with it came new ideas for exchanging dynamic information. Web 2.0 allows developers to create new dynamic web applications that utilize services and data stored in back-end databases.

Websites with back-end databases are often susceptible to web attacks, in particular Structured Query Language SQL Injection Attacks (SQLIAs). Over the past 15 years SQLIAs have been actively studied. As a result various approaches to solutions have been proposed; however, these approaches have yet to successfully solve the problem of combating SQLIAs.

Since 2004, the Open Web Application Security Project Open Web Application Security Project (OWASP) [2, 4] has published the Top 10 list of web vulnerabilities every three years. In 2004, SQL injection was number six, in 2007, it was number two, and since 2010 it remained the number one security risk facing organizations as it relates to web applications.

Seventeen years ago, Web 1.0 statically linked web pages were typically created by experienced web developers [5]. As Internet usage grew the demand for more dynamic content also grew. The explosive growth in the demand for websites caused a transition from experienced developers to an expanded base of developers with limited knowledge of programming or secure development techniques [6]. Currently, web pages no longer contain just statically linked content that seldom changes. Most modern web pages contain fully interactive user experiences, with high levels of user interaction and dynamic data hosted by back end database management systems Database Management Systems (DBMS).

1.2 The Proposed Approach and Solution: Lest Privilege Design and Semi-Automated Refactoring

This dissertation presents a systematic method and associated tool-set for protecting web applications. The proposed solution for securing an existing web application is performed in three phases. In the first phase, the non-least privilege behavior of the web application is learned and modeled. In the second phase, the web application is automatically converted to a least privilege model based on the functionality learned in the first phase. In the third phase, the new web application is evaluated in two different case studies against different SQL injection attacks. Figure 1.1 illustrates the flow of the proposed solution. It is a continual flow that checks and then enforces a least privilege model.

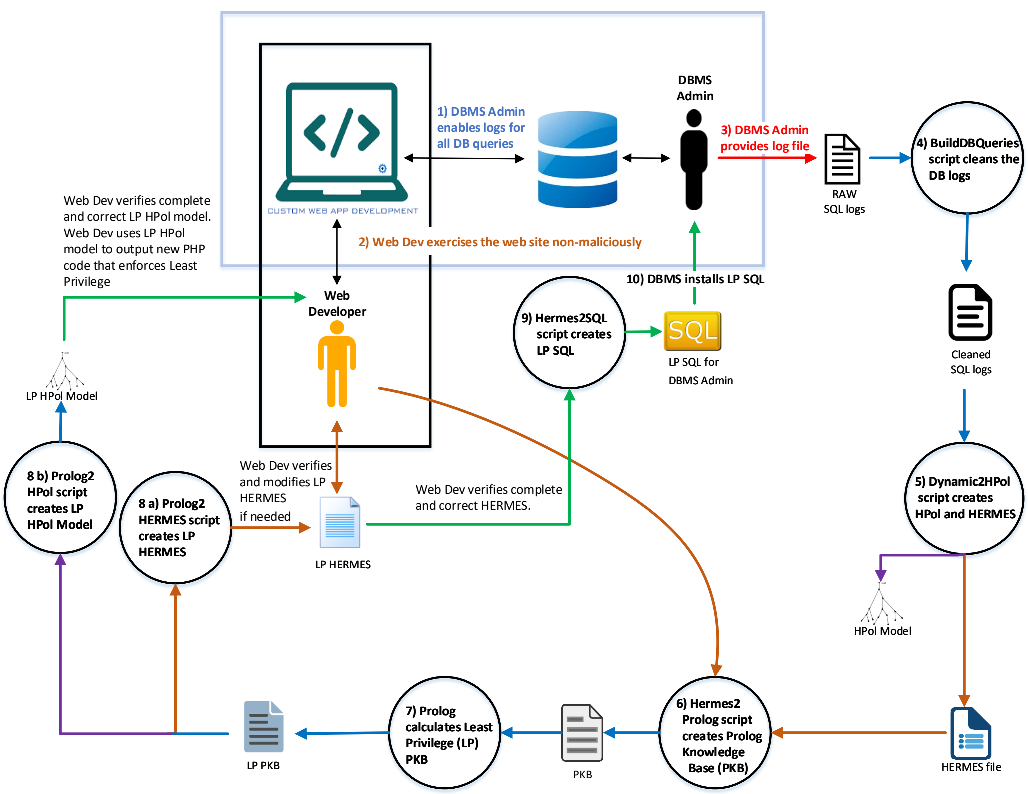


Figure 1.1: Architectural Overview: The complete architectural overview and the six contributions described in this dissertation.

1.3 Objectives of this Dissertation

The goal of this dissertation is to answer the primary question: **How can the large number of existing web applications be secured with limited resources and without manually rewriting millions of lines of code?** The primary question was divided into the following research objectives.

- **Objective 1**: Identify the current research on SQLIA from the ACM digital library and the IEEE Explore digital library.
- **Objective 2**: Evaluate the current research on SQLIA mitigation techniques.
- **Objective 3**: Evaluate which technologies and programming languages are currently used to develop dynamic web applications.
- **Objective 4**: Evaluate the applicability and fitness of the Hierarchical Policy Model (HPol) and the High-Level Easily Reconfigurable Machine Environment Specification (HERMES) approaches for modeling web application security.
- **Objective 5**: Develop a novel method for securing web applications using the principle of least privilege and the HPol formal security model.
- **Objective 6**: Develop a novel method of semi-automatically securing web applications using a least privilege approach.
- **Objective 7**: Evaluate the performance of this new method for mitigating SQL injection attacks using two case study web applications.

In summary the goal of this research is to provide a semi-automated process to secure a web application using the principle of least privilege, and with a minimal need to manually modify the web application source code.

1.4 Contributions of this Dissertation

To provide a semantic least privilege semi-automated approach to preventing cyber attacks on web applications the following contributions were completed as part of this dissertation.

- **Contribution 1: A manual but formal, systematic and repeatable process for securing current web applications based on the principle of least privilege.**

  Once the appropriate HPol model was created for web applications, that model was manually applied to two different case web applications as case studies. The developed model follows the typical Subject, Action, Object structure that all HPol formal models follow. These details are described in Chapter 4.

- **Contribution 2: Formal web application security policy modeling**

  I developed enhancements to HPol that enabled HPol to model web application security and access control policies. I applied these enhancements to enforce the principle of least privilege in web applications. This contribution is described in Chapter 5.

- **Contribution 3: Formal High-Level Easily Reconfigurable Specification.**

  The HERMES specification language was enhanced to enable it to represent web application security models. In addition, I developed the formal grammar and a new updated parser that supports the latest language improvements. The formal grammar and specifications for HERMES were created in order to ensure a human readable easily reconfigurable high-level specification that could be used by a DBMS administrator. HPol was modularized and new first order predicates were added. The details of HERMES and the enhancements to HPol are described in Chapter 5.

- **Contribution 4: Developed an approach and associated tools for automatically learning the database-level permissions needed on the database management system for a web application to operate with the least privilege possible.**

  SQL logs are the record of the transactions between the web pages, the database user, the database queries and the database tables. I conducted and logged non-attack database queries. Based on the logs an inference of the non-least privilege model was created. Chapter 6 explains the process of automatically creating the security model of the web application by using the SQL logs.

- **Contribution 5: Developed a formal, repeatable, and automated approach and associated toolset for determining and applying least privilege permissions at the database level for securing web applications.**

  The next step in the process was to automatically enforce the principle of least privilege on the web application. Chapter 7 explains the process of creating the least privilege model, and then enforcing the least privilege model on the web application via limited SQL users with limited permissions. Once the SQL users were updated the web application was attacked and the results were documented. The results are illustrated by two different case studies using educational real world web applications.

- **Contribution 6: Developed a systematic process for PHP code modification to assist the web developer in applying least privilege permissions for securing web applications.**

  The final step in the process was to develop a systematic step-by-step process for enforcing the principle of least privilege on the web application. Chapter 8 explains the step-by-step process of modifying the PHP code needed to enforce the least privilege model on the web application via utilization of the limited SQL users with limited permissions. The results are illustrated by two different case studies using educational real world web applications.

This dissertation provides a least privilege semi-automated approach to preventing cyber attacks on web applications. This work provides a formal, repeatable, and semi-automated approach and associated toolset for determining and applying least privilege permissions at the database level.

1.5 Scope of Achieved Mitigation and Defense

This section provides an understanding of the SQL injection attacks and how each attack is mitigated by the work of this dissertation. The following summarizes the types of attacks, and how this dissertation may mitigate the attack. An example of each query and attack are represented below with the attack portion of the query represented as blue text. Information concerning SQL injection can be found on the OWASP web site at https://www.owasp.org/index.php/SQL_Injection [7].

- **Tautology Attacks** - These injections allow unauthorized access to the database by ensuring the final query always returns true.

  ```
  SELECT * FROM user WHERE uname = ''OR 1=1 -- ...
  ```

  This approach does not mitigate the simple tautology attack, because the simple tautology attack is restricted to the same permissions on the same table. If a more advanced query is stacked with a simple tautology attack, and that attack references a different set of tables, then that attack would be mitigated by our work. If the attack uses the same tables originally used in the query then the attack will not be mitigated. If the attack uses any other tables or any other databases then the attack will be mitigated. If the attack uses special functions, or procedures, or read and write from file, because most pages don't need that access, then those attacks will also be mitigated.

- **Logically Incorrect Attacks** also called **Blind Attacks** - These injections are intended to trigger errors in the database, in such a way that the malicious actor can gather information about the database.

```
SELECT * FROM user WHERE uname = '11' AND password = '123' AND
CONVERT(int,(SELECT name FROM system WHERE type = 'u')); -- ...
```

  The research presented in this dissertation will mitigate most of these types of attacks, because these attacks usually need to execute database functions, which under a least privilege scenario, most pages are not granted access to database functions.

- **Union Attacks** - These queries contain separate queries, where each query is executed separately, and the results of each query are then combined using the keyword `UNION`. These types of attacks are intended to subvert the original query to add to the result sets the results of another query. Union queries can be utilized to bypass the restriction of stacked queries. Union queries are restricted to the select statement; while stacked queries are not.

```
SELECT id FROM user WHERE uname='admin
UNION SELECT id FROM db.table WHERE uname='admin'; '-- ...
```

  The work in this dissertation possibly mitigates this attack. If the attack uses the same tables originally used in the query then the attack will not be mitigated. If the attack uses any other tables or any other databases then the attack will be mitigated. If the attack uses special functions, or procedures, or read and write from file, because most pages don't need that access, then those attacks will also be mitigated. An example of this attack and the resulting mitigation is illustrated in Section 9.8.

- **Stored Procedures Attacks** - A stored procedure attack attempts to create stored procedures or functions.

  ```
  CREATE PROCEDURE DB @uname, @passwd, AS EXEC (SELECT * FROM user WHERE
  id= "'+@uname+"' and password = "'+@passwd+"'); GO
  ```

  The work in this dissertation will prevent these attacks. In general it is extremely rare that any web application middleware will need permissions to create stored procedures. Creating stored procedures should only be left to the database administrator. This work should prevent any creation of stored procedures. Execution of stored procedures should be mitigated for which the page doesn't have permissions. The type of defense for stored procedure attacks is similar to the type of defense of logically incorrect attacks. In MySQL there is no distinction between procedures and functions.

- **Piggy-Backed Attacks** also called **Stacked Queries Attacks** This attack attaches a separate, different malicious query to the existing query by adding a semicolon at the end of the original query.

  ```
  SELECT * FROM user WHERE uname = '111' and passwd = 'abc'; DROP TABLE
  user;
  ```

  The work in this dissertation most likely mitigates this attack. If the attack uses the same tables originally used in the query then the attack will not be mitigated. If the attack uses any other tables or any other databases then the attack will be mitigated. If the attack uses special functions, or procedures, or read and write from file, because most pages don't need that access, then those attacks will also be mitigated. An example of this attack and the resulting mitigation is illustrated in Section 9.4.

- **Inference Attacks** - An inference attack is an SQL injection containing a conditional construct. It uses a specific instruction, such as time delay, to trigger noticeable database behavior. This type attack allows the malicious actor to infer if the tested expression was true or false.

```
DECLARE @s varchar(8000) SELECT @s = db_name() IF (ascii(substring(@s, 1,
    1)) & ( power(2, 0))) > 0 WAITFOR delay '0:0:5'
```

  The work in this dissertation should mitigate most of these attacks. If the attack uses the same tables originally used in the query then the attack will not be mitigated. If the attack uses any other tables or any other databases then the attack will be mitigated. If the attack uses special functions, or procedures, or read and write from file, because most pages don't need that access, then those attacks will also be mitigated.

- **Alternate encoding** - Encode attacks in such a way to avoid standard input filtering. The original query of `SELECT info FROM user WHERE login='login' AND pin='pin';` [8] can be represented with the input of a 0 for the pin as

```
"0; DECLARE @a char(20) SELECT @a=0x73687574646f776e EXEC(@a)"
```

  Alternate encoding attacks are considered the most advanced attacks for SQL injection. If the attack uses the same tables originally used in the query then the attack will not be mitigated. If the attack uses any other tables or any other databases then the attack will be mitigated. If the attack uses special functions, or procedures, or read and write from file, because most pages don't need that access, then those attacks will also be mitigated.

1.6 Organization of this Dissertation

The remainder of this dissertation is organized as follows.

Chapter 2 explains the background of cyber attacks on a web application. Part of understanding cyber attacks includes an explanation of web application security policies including a basic understanding of SQL injection and the Principle Of Least Privilege The Principle of Least Privilege (POLP). A formal policy model and a formal high-level easily reconfigurable specification are introduced.

Chapter 3 explains the current state-of-the-art solutions in web application security. This chapter includes a formal concept analysis model of different runtime solutions. This chapter also includes an analysis of the current static solutions.

Chapter 4 explains how the POLP was manually applied to a case study web application. This chapter contains the details for Contribution 1.

Chapter 5 explains the enhancements to HPol and HERMES. These enhancements allow for future work to revisit previous HPol work, unrelated to this dissertation. Contribution 2 and 3 are discussed in this chapter.

Chapter 6 explains the automated process to dynamically build a non-least privilege security model. This involves running the web application without injection attacks. A baseline was developed by running the web application attack free, and identifying the users, the pages executing database commands, and the database tables being accessed. This chapter contains the details for Contribution 4.

Chapter 7 explains the process to dynamically build the new Least Privilege HPol model. As part of the automation process SQL grant statements are created. The web application administrator executes the provided SQL statements. These statements are used to enforce least privilege. This chapter explains the details of Contribution 5.

Chapter 8 explains the process to refactor the PHP code to fully move the web application from non-least privilege to least privilege. This chapter contains the details for Contribution 6.

Chapter 9 explains the process of refactoring the SEED Labs web application. The details of Contribution 6 are explained in this chapter.

Chapter 10 explains the process of refactoring the Mutillidae II web application (Mutillidae). This chapter also contains details for Contribution 6.

Chapter 11 discusses the related works that are also mitigating similar problems. This chapter describes some of the limitations of trying to solve SQL injection without sanitization of inputs.

Chapter 12 summarizes and concludes the work and contributions completed in this dissertation. This chapter discusses the assumptions and limitations, threats to validity and potential future work.

Chapter 2: Background

This chapter examines web security, web application cyber attacks, including the concept of SQL injection attacks, and the concept of the principle of least privilege. This chapter also explains the security model for web applications via the Hierarchical Policy Model. The High-Level Easily Reconfigurable Machine Environment Specification is also introduced as a mechanism to easily modify the security policy model.

2.1 Cyber Attack Definitions and Security Policies

Since SQL injection first appeared on the OWASP Top 10 list in 2006 there have been more than 400 papers written about vulnerability and attack type classification systems and mitigation techniques. SQLIAs occur in various ways; however, SQLIAs most commonly occur when malicious user-provided data is passed through the web application as SQL commands, and is executed as SQL code by the backend database. In 2006 Halfond et al. [8] characterized seven types of SQLIAs, based on the goal and the intent of the attacker. Those seven types are tautologies, incorrect queries, union query, piggy-backed queries, stored procedures, inference, and alternate encoding.

Others have extended Halfond's et al. work. Since SQLIAs are initiated through a web page, in 2009 Seixas et al. [9] identified and classified the most common security vulnerabilities in web programming languages. In 2012 Ray and Ligatti [10] formally defined web applications and code-injection attacks. In 2013 Shar and Tan [11] broadly classified SQL injection defenses into three categories, defensive coding, SQL injection vulnerabilities detection, and SQLIA runtime prevention. Shar and Tan [11] state, *"The best strategy for combating SQL injection . . . calls for integrating defensive coding practices with both vulnerability detection and runtime attack prevention methods."* Defensive coding practices are important; however, most developers typically have less than five years of professional experience (57%) [6]. Furthermore, of the hundreds of different SQL injection research mitigation techniques, no technique has technologically transferred to enterprise use.

Teaching secure development techniques to the next generation of web developers is very important; however, implementing secure development practices will potentially take years. An immediate solution is needed, specifically a systematic and semi-automated least privilege solution, that helps today's developers secure today's web applications.

2.2 The Principle of Least Privilege

A major root problem contributing to vulnerabilities in web applications is the widespread usage of the highest privilege design pattern. In such a pattern, users and applications are given the highest level of privileges needed to execute the union of all needed tasks. In the case of web applications, this design pattern translates into the practice of granting the web application and/or middleware processes root privileges over the back-end database management system (DBMS). The same pattern can also used to grant the middleware processes root-level privileges over the file system within the web application server.

Because of the problem described above, once a web application has been compromised an attacker can easily gain root-level access to the back-end database. Using such a design pattern violates two of the most basic principles of secure system design: (1) Least Privilege and (2) Layering or Defense in Depth [12].

2.3 The Hierarchical Policy Model (HPol)

The Hierarchical Policy (HPol) formal model enables the representation of access control and security policies using a hierarchical graph structure. In HPol, subjects, actions, and objects are represented by a hierarchical graph. Policies are represented by links, or relation tuples, which state that a given subject has been granted permission to perform a given action on a given subject. HPol represents each subject, action, and object with a node within a directed acyclic graph (DAG). Policy links between nodes connect subject, action, object nodes within the DAG to indicate what policies are allowed [13].

Figure 2.1 shows a portion of the resulting HPol model for the highest privilege design of the Mutillidae II web application (Mutillidae) [14]. In the model the user `dbRoot` is granted administrative-level permissions for all objects within all databases within the DBMS.

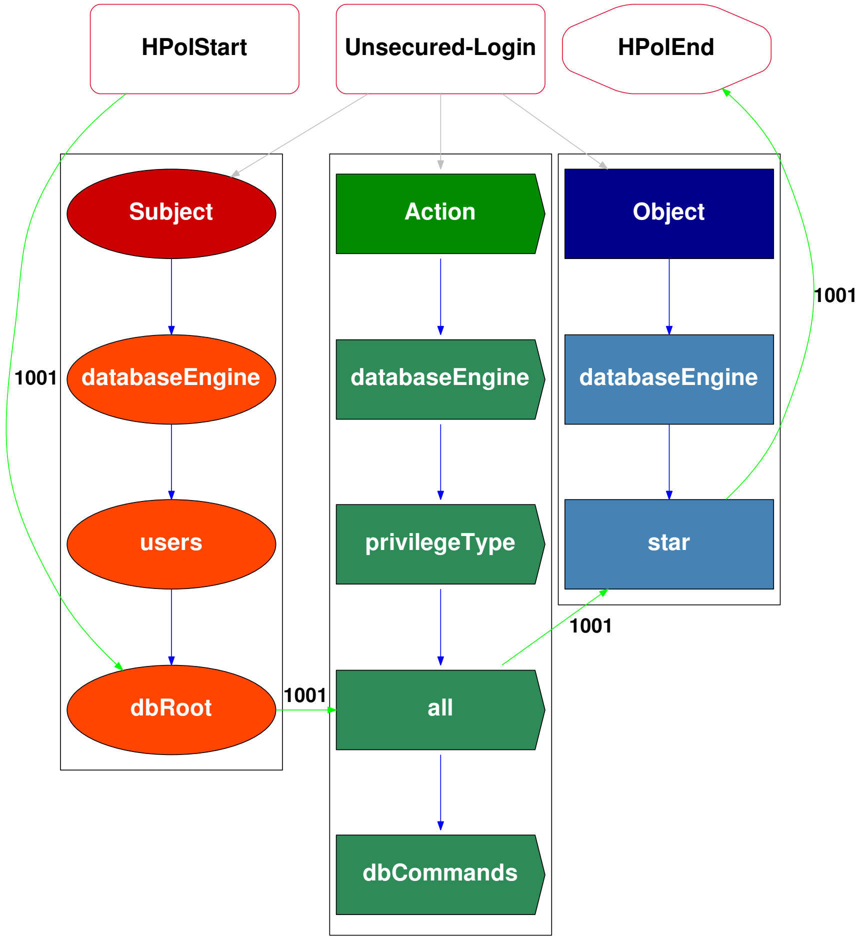


Figure 2.1: HPol Non-Least Privilege: An HPol example illustrating a non-least privilege database user dbRoot. The dbRoot user has full access to the entire DBMS.

Policy number 1001 indicates the granting of such permissions to the corresponding subject, action, and object tuple.

2.4 High-Level Easily Reconfigurable Machine Environment Specification (HERMES)

The High-Level Easily Reconfigurable Machine Environment Specification (HERMES) enables modern organizations to design and implement highly-specific tailored security configurations. HERMES is an enterprise-wide and policy-oriented, rather than configuration-oriented, security configuration management system. HERMES is a high-level security policy description language

```
Node: Example
{
      FQN: Example.Example.Example;
      Description: "HPol Root Node";
      Path: "Example";
      Type: "HPolRoot";
}.

Node: Subjects
{
      FQN: Example.Example.Example.Subjects;
      Description: "Subjects";
      Path: "Example/Subjects";
      Type: "Subjects";
}.

Node: databaseEngine
{
      FQN: Example.Example.Example.Subjects.databaseEngine;
      Description: "databaseEngine";
      Path: "Example/Subjects/databaseEngine";
      Type: "subject";
}.

Node: users
{
      FQN: Example.Example.Example.Subjects.databaseEngine.users;
      Description: "users";
      Path: "Example/Subjects/databaseEngine/users";
      Type: "subject";
}.

Node: dbRoot
{
      FQN: Example.Example.Example.Subjects.databaseEngine.users.dbRoot;
      Description: "dbRoot";
      Path: "Example/Subjects/databaseEngine/users/dbRoot";
      Type: "subject";
}.

Policy: PID_1001
{
      FQN: Example.Example.PID_1001;
      Description: 'Policy';
      Status: ENABLED;
      AbsolutePath: [HPolStart,
            Example.Subjects.databaseEngine.users.dbRoot,
            Example.Actions.databaseEngine.privilegeType.all,
            Example.Objects.databaseEngine.star,
            HPolEnd];
      RelativePath: [HPolStart, dbRoot, all, star, HPolEnd];
}.
```

Figure 2.2: HERMES: A HERMES specification example illustrating the HPol model represented by Figure 2.1, which illustrates that a non-least privilege database user dbRoot that has full access to the entire database system.

that enables system administrators to write security policies that can then be implemented across the IT/OT ecosystem [15].

HERMES allows IT/OT security personnel to describe their organization and security policies based on the description of two domains: Nodes and Policies. Each of these domains can be defined using a hierarchical structure. Nodes encompass organizational domains and devices, groups of users and roles, applications, and actions. Policies are declared as actions applied to a given combination of nodes. HERMES is used to easily convert a HPol security model from the DAG to Prolog, and from Prolog back to a HPol security model. HERMES also allows for easy and simple additions or changes to the current structure. Simply stated HERMES is the high-level text representation of a HPol security model.

Figure 2.2 shows a portion of the resulting HERMES language for the highest privilege design of the Mutillidae web application from Figure 2.1. In the specification policy, identification number 1001 (PID_1001) specifies the user `dbRoot` is granted administrative-level permissions for all objects (symbolized as `star` meaning all) within all databases within the DBMS.

2.5 Formal Concept Analysis (FCA)

Formal Concept Analysis (FCA) is "*. . . a mathematical formalism which analyses the data in a context and attempts to extract the concepts embodied within that data* [16]." FCA is a method for creating a context hierarchy from a collection of objects and their properties. Each concept in the hierarchy represents the set of objects sharing the same values for a certain set of properties or attributes. A hierarchy is a mathematical concept where a set is ordered. For example, the set of integers is one such mathematical hierarchy. For FCA, the ordered set is determined by all objects belonging to a concept, and by the collection of all attributes shared by the object [17].

Context [16] is the triplet (G, M, I) where G represents the objects, M represents the attributes, and I represents the relationship of objects to the attributes $I \subseteq (G \times M)$. Content is a pair of sets defined as $(A \subseteq G, B \subseteq M)$, where A is the set of all objects that have all the attributes in B. B represents the set of all attributes that apply to all objects in A.

The context of objects and attributes is constructed as a two-dimensional array of binary values representing the binary relations between the objects and the attributes. Table 2.1 is a simple example illustrating the FCA of animals (objects) and the locations where the animals live (attributes). The rows represent the objects (animals) and the columns represent attributes (the locations).

Table 2.1: Formal Concept Analysis example illustrating object/attributes relationship

Object Name	Land	Water	Trees
Humans	✓		
Frogs	✓	✓	
Monkeys	✓		✓
Giraffes	✓		
Fish		✓	
Turtles	✓	✓	

Concepts are understood as *". . . the basic units of thought formed in dynamic processes within social and cultural environments [17]."* Concepts and concept hierarchies, are used to create a mathematical model that allows a lexical relationship between objects, attributes, and the relationships of the objects to the attributes. The lexical relationship indicates that an object has an attribute. A lattice is created using the relationship between objects and attributes and the theory of concepts, which is rooted in philosophy and psychology. Although, there are twelve aspects to the theory of concepts, those aspects can be summarized in the following statements.

- The mathematical notion of a formal context converts to the logical meaning of a domain of interest based on object-attribute-relationships.
- The mathematical order-relationship that a formal concept is less than another formal concept is logically understood as a subconcept-superconcept-relationship.
- The mathematical derivation of a set of formal attributes is logically viewed as the identification of all objects having all attributes of a given attribute collection.

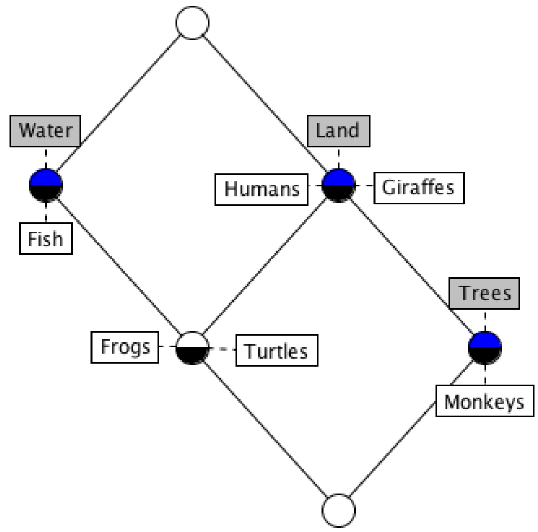


Figure 2.3: FCA Lattice: The FCA lattice corresponding to Table 2.1

- The labeled line diagram of a concept lattice is logically considered as a hierarchical network linking nodes with object names to nodes with attribute names and thereby establishing conceptual meanings.

- Formal object and attribute implications lead to the recognition of conceptual dependencies within the given domain of interest.

The lattice [17] is a visual representation of all objects that share the given attributes, and the relationship of all attributes shared by the given objects. Figure 2.3 illustrates the lattice generated by the set of concepts illustrated in Table 2.1.

Structured analysis is defined as the process of studying a system in order to identify the system's structure, goals and purposes, as well as create systems and procedures that will achieve the structure, goals and purposes in an efficient manner. As previous stated Formal Concept Analysis is "*. . . a mathematical formalism which analyses the data in a context and attempts to extract the concepts embodied within that data* [16]." FCA is a method for creating a context hierarchy that identifies the system's structure.

FCA was utilized as a means to apply mathematical representations to runtime mitigation techniques. By using a FCA it was possible to identify a solution for runtime mitigation techniques that could used to mitigate SQLIAs with minimal programmer involvement.

Chapter 3: Investigation of the Current State of the Art in Web Application Security

This chapter presents an overview of the current state-of-the-art techniques in web application security. These techniques involve different mechanisms to mitigate SQL injection, including dynamic techniques, static techniques, and hybrid techniques. This chapter also explains how and why SQL injection remains on the OWASP Top 10 list [2]. This chapter further defines the common dynamic techniques including categorizing those techniques as the most promising mitigation techniques via in-depth analysis known as Formal Concept Analysis. This chapter also further describes static techniques and hybrid techniques.

3.1 Current State of Web Application Security

The current state-of-the-art techniques for mitigating web application vulnerabilities and attacks, specifically SQL injection, can be categorized into one of three areas:

1. Dynamic solutions attempt to detect and possibly handle the SQL injection attack during the runtime of the web application.

2. Static solutions scan the web application source code or the web application structure to identify all possible locations where the source code or structure could be vulnerable to SQL injection.

3. Hybrid solutions are a combination of static and dynamic solutions.

Dynamic solutions can be categorized as string analysis approaches. String analysis approaches typically involve data input searching or filtering for malicious SQL keywords that can be used to attack the database. If the data inputs are detected as compromised, the web application does not pass the query string to the database engine. A key problem with string analysis approaches is the extreme difficulty in detecting all data input variations related to SQL injection attacks.

Static solutions can be categorized as black box analysis or white box analysis. Black box analysis first involves identifying weak points in the web application by using a web crawler to detect the application's workflow, including vulnerable points. Second the black box test generates an attack for one of the vulnerable points. As the attack occurs, the black box test monitors the behavior of the application to determine if the attack was successful. White box analysis is based on examining the web application's source code and its structure to detect the vulnerable points.

Hybrid solutions involve both dynamic and static solutions. A hybrid solution typically involves black box analysis as the static solution. The black box test identifies the vulnerabilities and then generates SQL injection attacks to test the vulnerability. The dynamic solution typically is then an input validation solution that attempts to sanitize the user inputs to prevent a SQL injection.

3.2 Why SQL Injection Attacks (SQLIAs) are Still an Unsolved Problem

Although SQL injection attacks have been discussed since 1998 [18], SQLIAs still remains on the OWASP Top 10 List [2] and it is still prevalent in many web applications. SQL injection may occur when the web application's backend web server and the database servers interact.

The two main interactions between the web application's backend web server and the database server are:

1. Directly entering database parameters into the URL string, typically via the HTTP GET command.
2. Data entered by the user into a web application form is passed to database server.

Considering modern highly interactive web application experiences there are dozens, if not hundreds of places across the web application that satisfy one of the above scenarios. The web is a set of intertwined web pages with intertwined technologies that is very complicated to ensure the web application works correctly across multiple devices. Therefore, SQL injection remains a problem for the following equally important reasons:

- The web is a complicated intertwined set of web pages with millions of lines of unvalidated and unsanitized code. Furthermore, most projects have budget and/or schedule constraints that prevent them from having adequate code reviews. Simply stated, it is not cost effective to manually rewrite millions of lines of code to properly secure it.

- There are a large number of developers that have less than five years of professional experience (57%) [6] and may not have knowledge of secure coding techniques. Part of the problem is the lack of adequate and cybersecurity focused online tutorials. Many of online tutorials either don't discuss SQL injection or the tutorial uses outdated or deprecated libraries. Furthermore, most tutorials illustrate a maximum privilege model instead of a least privilege model.

- Open-source and free tools that can be applied to the millions of lines of insecure code and be used without developer interaction have only been developed in research and have not transitioned to enterprise use. What is urgently needed is an environment that requires little to no programmer involvement, yet still detects a majority of SQLIAs. This type of approach has already been implemented with other security vulnerabilities. A good example is the mitigation of most, but not all, buffer overflow attacks with tools such as StackGuard or Address Space Layout Randomization (ASLR), where both techniques helped raise the security barrier, without requiring major code modifications.

- Most web applications contain a database with user demographics, possibly including credit card numbers, birthdays and other sensitive information. This data is know as "a juicy target;" once it has been retrieved from the database it can be sold and used maliciously.

- With the surge of fully interactive web applications it is difficult for experienced developers to become and stay current on the new technologies. With the increased use of JavaScript, a new set of vulnerabilities have been introduced.

Almost 20 years later SQL injection still remains a problem for the reasons stated above. The research presented here develops a systematic, semi-automated, formal and repeatable system which is effective at modeling the security polices of web applications as well as preventing attacks. Furthermore, the system can be utilized by web application developers regardless of experience.

3.3 Runtime Mitigation Techniques for SQLIAs

Previous research into SQLIA mitigation techniques, broadly classified runtime mitigation techniques as SQLIA Detection (SQLIAD) or SQLIA Prevention (SQLIAP). Halfond et al. [8] define detection techniques as *" . . techniques that detect attacks mostly at runtime."* This section summarizes some of the most popular runtime SQLIA Detection mitigation techniques. Table 3.1 details the 10 runtime techniques that require little to no programmer involvement, yet still detect a majority of SQLIAs. These techniques were chosen based on the following criteria:

- Number of citations
- Limited programmer involvement
- Limited additional resources
- Considered to be only runtime techniques
- Number of times a technique is mentioned in previous survey papers [8, 11, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33] (See Section 11.5 for a discussion of these papers.)

Furthermore, identifying which mitigation techniques and characteristics were important for SQLIADs Formal Concept Analysis, a case study of similar types of security vulnerabilities was conducted. The most closely related vulnerability is buffer overflow attack mitigation techniques. Similar to SQLIADs, buffer overflow attacks can easily be mitigated by the developer through secure development techniques. Unlike buffer overflow mitigation techniques SQLIADs techniques have not successfully transitioned to enterprise use.

Table 3.1: Summary of SQL injection runtime mitigation techniques

Technique	Brief Technique Summary
CANDID [34]	CANDID combats SQL injection via obfuscation and de-obfuscation of SQL commands. SQL injection attacks can be detected based on dynamic verification performed on the obfuscated queries.
CSSE [35]	Context-Sensitive String Evaluation uses a modified PHP interpreter to track precise per-character taint information through the system. A context sensitive analysis is used to detect and reject queries.
SQLCheck [36]	SQLCheck checks queries at runtime for conformity to a model of expected queries. The model is expressed as a grammar that will only accept legal queries. In order to check queries at runtime a key is used to delimit user input.
SQLGuard [37]	SQLGuard compares the parse tree of the SQL statement before inclusion of user input with the SQL statement after inclusion of user input. The developer must use a special library.
SQLProb [38]	SQLProb extracts user query inputs with a pairwise alignment algorithm to compare user queries against legitimate queries. It then uses a SQL parser to check each extracted input. The query is only sent to the database if the user input is syntactically confined.
SQLrand [39]	SQLrand provides a proxy server between the web server and the database server which is used to decipher the received queries. The proxy server un-randomizes the SQL queries and then forwards the converted SQL query to the database server for execution. It also hides any database server error messages.
WASP [40]	WASP uses positive tainting, and tracking techniques for syntax-aware evaluation of queries string.
Header Sanitization [41]	Header Sanitization sanitizes received variables inside HTTP header request methods. The sanitized content is replaced back into the original header field.
Network Analyzer [42]	Network Analyzer builds a detection system between the attacker and the web server. This system analyzes headers and payload via “Deep Packet Inspection” of the packet.
Web Application Firewall [43]	Web Application analyzes the received variables inside HTTP header request methods. The sanitized content is either rejected or passed to the SQL engine if no SQL injection is detected.

3.4 Structured Analysis of Runtime Mitigations for SQLIAs

The attributes for the FCA were chosen from a combination of attributes in prior survey papers and criteria identified in selecting the 10 mitigation techniques. Not all attributes were chosen from prior survey papers since those attributes did not add value to the FCA. For example a prior survey paper classified the SQLIAD mitigation techniques based on the ability to generate test suites for attacks. Since only SQLIA runtime techniques are being considered, the ability to generate test suites was not considered during attributes selection. To be considered an attribute from a prior survey paper the attribute had to appear in at least four survey papers.

The attributes chosen from the most commonly discussed in survey papers were:

- Classification of the technique as SQL detection
- Code base modification
- Additional infrastructure requirements
- The types of SQL injection attacks for which the technique would mitigate

Recall the focus of the structured analysis and FCA is to identify which SQL injection runtime mitigation techniques would be the best candidates for preventing SQLIAs with minimal developer involvement; therefore, additional attributes needed to be identified. The identified additional attributes that did not appear in any prior survey papers included:

- Was the technique language specific?
- Did the implementation require tracking or tainting?
- Did the implementation utilize the GET or POST references?
- Was the technique open source?
- Was the technique in active development?
- What was the level of required developer involvement?

Table 3.2: Formal Concept Analysis of SQL injection mitigation techniques

Mitigation Name	CANDID	CSSE	SQLCheck	SQLGuard	SQLProb	SQLrand	WASP	Header Sanitization	Network Analyzer	Web App Firewall
Low Programmer Involvement	✓				✓		✓			
High Programmer Involvement		✓	✓	✓		✓		✓		
Automatic Code Modification			✓						✓	
Manual Code Modification					✓		✓			
No Code Modification	✓	✓		✓		✓		✓		
Additional Infrastructure	✓	✓			✓			✓		
Open Source	✓	✓	✓	✓	✓	✓		✓		
Language Specific	✓	✓		✓		✓		✓		✓
Active Development						✓	✓	✓	✓	
Tracking		✓					✓			
GET/POST		✓						✓	✓	✓
Tautology *	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Logically Incorrect *	✓	✓	✓	✓		✓	✓	✓	✓	
Union Query *			✓	✓	✓	✓	✓		✓	✓
Stored Procedures *				✓						
Piggy-Back Query *		✓	✓	✓	✓	✓				
Inference *		✓	✓	✓	✓	✓	✓			
Alternative Encodings *		✓	✓	✓		✓				

In order to be classified as a runtime technique a majority of all attributes must be met. These SQL injection attack types were classified as very important because the ultimate goal is to mitigate SQL injection attacks with minimal developer involvement. Table 3.2 illustrates the relationship between the mitigation techniques (objects) and the analysis of each technique (attributes). The attributes includes a mapping of the mitigation techniques to the types of SQL injection and are denoted by the asterisk (*).

The free open source software Concept Explorer developed by Yevtushenko [44] was used to construct a two-dimensional array structure. This two-dimensional array structure is a binary structure represented with ones and zeros, a zero is represented by a blank and a one is represented by a check mark. The objects appear as individual rows and the columns are the individual attributes. The illustration of the objects and attributes is illustrated in Table 3.2.

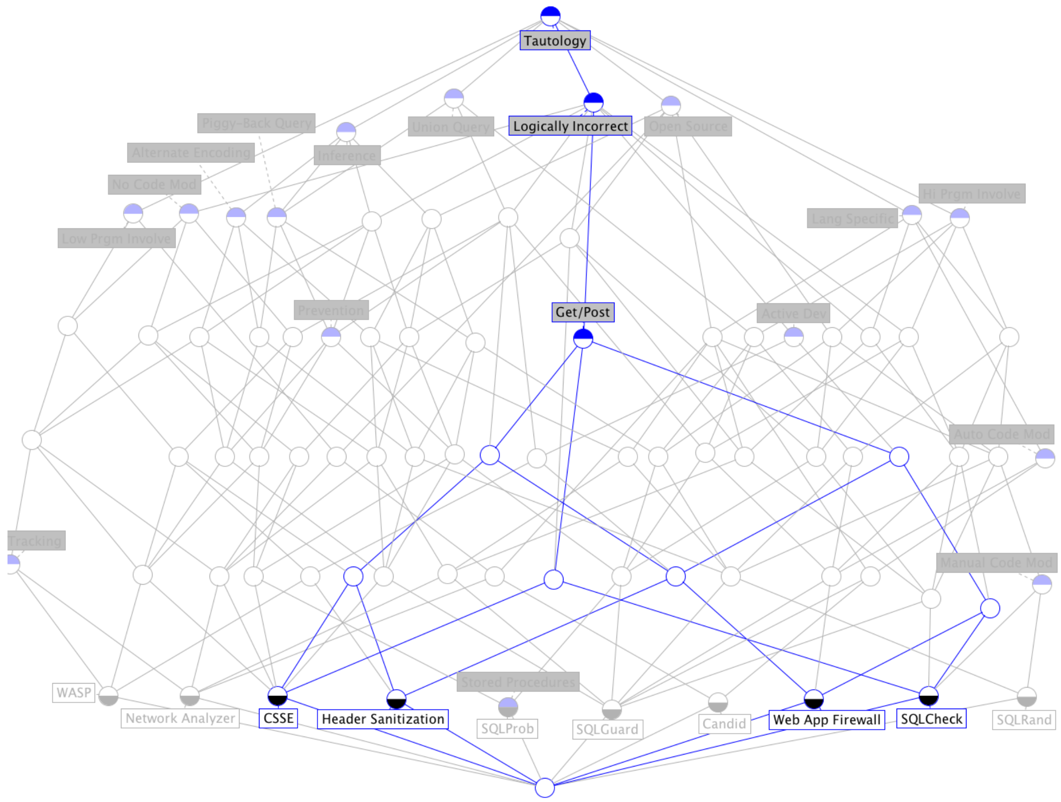


Figure 3.1: FCA Lattice: The Formal Concept Analysis lattice that illustrates the relationship for all objects that contain the attribute "GET/POST", generated by Concept Explorer [44].

A concept lattice is uniquely determined by its formal context, meaning every structural property can be read based on the incidence relation. An incidence relation is defined as the binary relationship between different types of objects, captured by the idea being expressed. For example "a point lies on a line" is an incidence relationship. For FCA lattices the binary relationship is derived from the binary table. Table 3.2 clearly illustrates the relationship expressed in the full lattice. Figure 3.1 illustrates the attributes and objects that contain the characteristic "GET/POST". Figure 3.2 illustrates the lattice derived for every structural property based on the incidence relation.

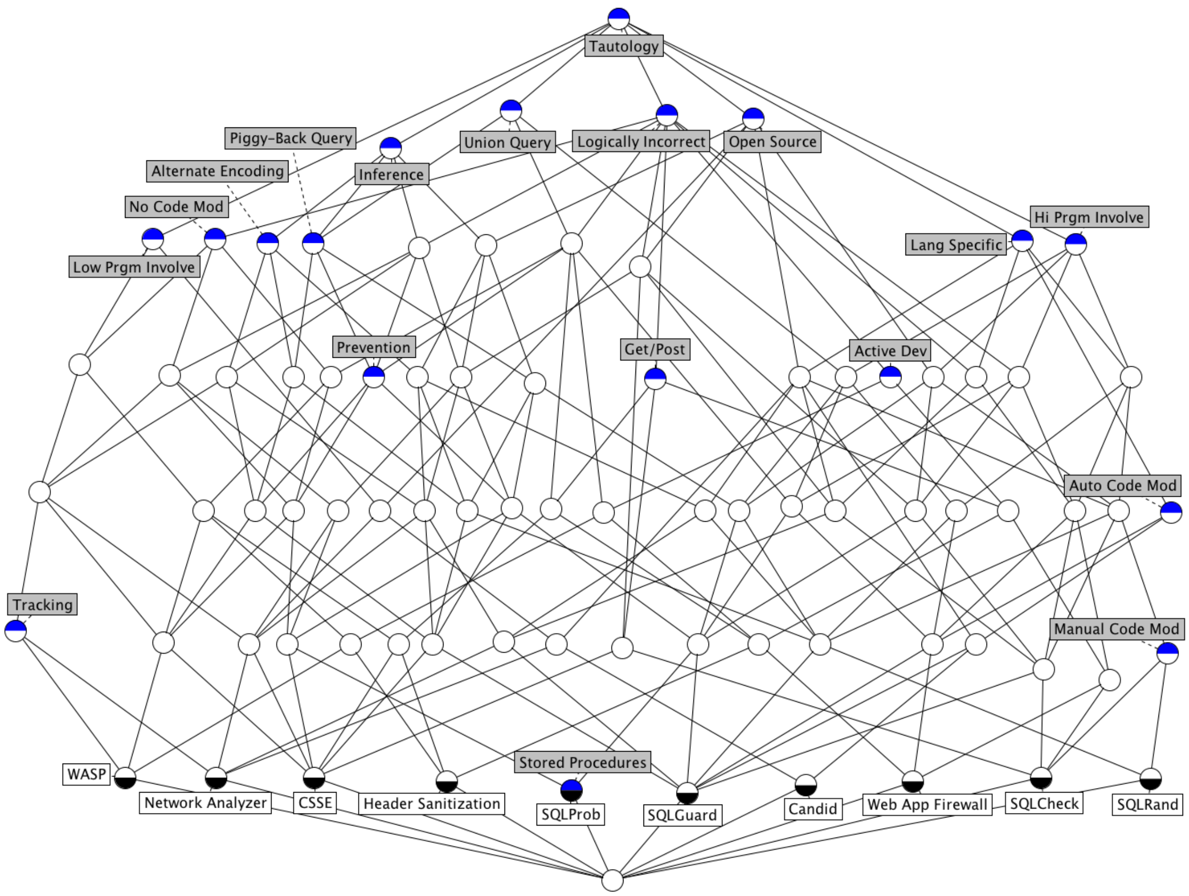


Figure 3.2: FFCA Lattice: The FCA lattice illustrating dynamic SQL injection mitigation techniques.

3.5 Static Mitigation Techniques for SQLIAs

Just as SQLIA Detection (SQLIAD) is for runtime mitigation techniques, then SQLIA Prevention (SQLIAP) is for static mitigation techniques. Halfond et al. [8] define prevention techniques as ". . . techniques that statically identify vulnerabilities in the code, propose a different development paradigm for applications that generate SQL queries, or add checks to the application to enforce defensive coding best practices."

Table 3.3: Summary of SQL injection static mitigation techniques

Technique	Brief Technique Summary
SQLUnitGen [45]	SQLUnitGen locates vulnerabilities though automated penetration testing, which generates test reports that require the developer to manually correct the vulnerability.
Ardilla [46]	Ardilla incorporates symbolic logic execution into randomized test inputs to identify previously undetected SQL injection vulnerabilities. Ardilla generates sample inputs, and creates attack vectors that are symbolically tracked as tainted inputs.
SAFELI [47]	SAFELI inspects Microsoft bytecode for an ASP.NET Web application, using symbolic execution. For each SQL query, a hybrid constraint solver is used to identify corresponding user inputs.
JDBC Checker [48]	JDBC Checker verifies the correctness of dynamically generated SELECT query strings. The string is analyzed via a Finite State Automata (FSA), that uses a framework to parse class files and compute inter-procedural control flow graphs.
TASA [49]	TASA is an ASP static analyzer that utilizes path-sensitive, inter-procedural and context-sensitive data flow analysis through taint propagation.
Pixy [50]	Pixy is a static analyzer that utilizes flow-sensitive, inter-procedural and context-sensitive data flow analysis to discover vulnerable points in the web application. Pixy targets general classes of taint-style vulnerabilities including SQL injection.
SecuBat [51]	SecuBat is a generic web vulnerability scanner, similar to a port scanner, that automatically analyzes web sites for exploitable SQL injection vulnerabilities.
PHP Static Detection [52]	PHP Static Detection captures information at decreasing levels of granularity at the intra-block, intra-procedural, and inter-procedural levels.

Prevention techniques can be further classified into two categories: white box (compile-time) analysis or black box (both static and runtime) analysis. For white box analysis the mitigation technique involves tools to examine the code and identify the potential SQL injection vulnerabilities. The developer needs to manually modify the vulnerable code to fix the vulnerability. Black box analysis requires an input generator that creates automated test cases. Those tests are executed against the existing code and are monitored during execution. The results of the execution are used to identify previously undiscovered vulnerabilities. Once the vulnerability is detected, correction requires developer involvement. Table 3.3 displays a brief summary of the eight static mitigation techniques.

The techniques were chosen based on the following criteria:

- Number of citations
- Considered to be a static technique (black box or white box analysis)
- Number of times a technique was mentioned in previous survey papers [21, 25, 53, 54, 55]

The techniques described in this dissertation differ from previous runtime and static techniques. Recall that runtime techniques detect attacks typically using string searching or string filtering. Static techniques involve scanning the source code for vulnerabilities. Based on the results of the static technique, detected code vulnerabilities are then modified before the web application is placed into production. This work is different from dynamic mitigation techniques in that this work does not detect attacks at runtime, nor does this work conduct string searching or string filtering. This work is different from static techniques in that this work does not scan the source code for vulnerabilities.

This work provides a least privilege implementation to SQL injection attacks. This solution infers the privilege model from from the SQL database transactions. It then dynamically builds the least privilege model and provides the system administrator the SQL statements to secure the database. This solution builds a systematic, semi-automated, formal and repeatable system that is effective at modeling the security polices of web applications as well as preventing attacks. This system can be implemented by developers, regardless of experience and knowledge of secure development practices.

Chapter 4: Manually Applying Least Privilege with HPol for Web Application Security

This chapter explains the Principle of Least Privilege (POLP) and how many web applications violate the POLP. Section 4.1 explains the process of manually applying least privilege. Section 4.2 explains the Hierarchical Policy Formal Security Model (HPol) and Section 4.3 presents High-Level Easily Reconfigurable Machine Environment Specification (HERMES). Using HPol and HERMES to build a formal web application security model is explained in Section 4.4. Using HPol and HERMES to build a formal web application filesystem security model is explained in Section 4.5 and a formal web application DBMS security model is explained in Section 4.6. With an understanding of the HPol formal security model and the specification outlined from HERMES, Section 4.7 discusses the application of HPol and HERMES to a case study of the Mutillidae II web application (Mutillidae). Figure 4.1 illustrates **Contribution 1: A manual but formal, systematic and repeatable process for securing current web applications based on the principle of least privilege**.

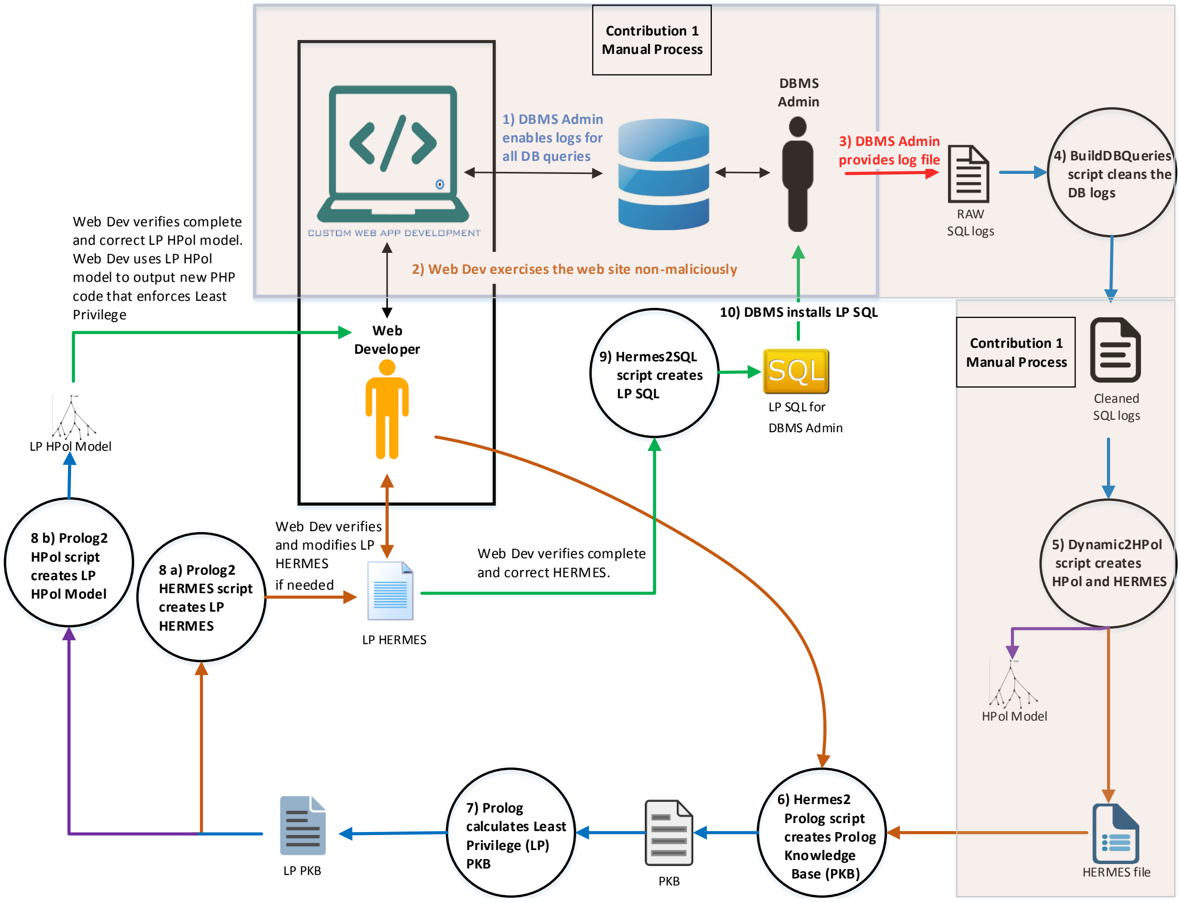


Figure 4.1: Contribution 1: A manual but formal, systematic and repeatable process for securing current web applications based on the principle of least privilege.

4.1 Manually Applying the Principle of Least Privilege

One of the primary reasons that many current Web applications contain security vulnerabilities is the widespread practice of highest privilege design or high watermark security policy. This widespread practice of highest privilege configuration entails giving the computing processes the highest level of privilege possible, instead of restricting the access permissions with the highest possible granularity and lesser privilege. For example, it is common practice today, that Web application middleware processes are given the highest level of access privilege (*root*) to the database management system (DBMS) containing the back-end database. Figure 4.2 illustrates the current practice for database configuration. This is an excerpt of standard available code tutorial for logging into a MySQL database from a PHP Web application. Observe that the

```
/**************************/
/* Database Configuration */
/**************************/
/* If there is any problem connecting,
   it is almost always one of these values. */
static public $mMySQLDatabaseHost = "localhost";

static public $mMySQLDatabaseUsername = "root";

static public $mMySQLDatabasePassword = "";

static public $mMySQLDatabaseName = "nowasp";
```

Figure 4.2: Current Practice Database Configuration: An excerpt of standard available code tutorial for logging into a MySQL database. (Mutillidae:2.6.42 GPL mutillidae/classes/MySQLHandler.php)

connection to the database is made with DBMS root privileges. If the Web Application is compromised then the attacker will have administrative-level privileges to all data within the DBMS.

The principle of least privilege (POLP) is a computer security concept that promotes minimal user privileges based on users' role. The POLP was originally described by Saltzer and Schroeder [12] to limit the potential damage of any security breach, whether accidental or malicious. The POLP should be applied to individual system components. Each system component should have the least authority necessary to perform the appropriate tasks. This helps reduce the vulnerabilities of the computer system by eliminating unnecessary privileges that can result in exploits and compromises [56, 57, 58, 59].

In the case of Web applications, for example, an index page that only displays basic information should not have write access to the file-system or the backend DBMS. Applying the POLP to the index page dictates the page should be read only and have no access to the backend DBMS. By correctly enforcing the POLP, an attack on the basic index page would mitigate unintended information disclosure. When adequately implementing least privilege, vulnerabilities in one part of an application are more difficult to leverage in order to gain access to other parts of the same system.

4.2 Manually Creating the HPol Formal Security Policy Model

The Hierarchical Policy (HPol) formal model, and its associated tool-set, was developed to enable a formal representation of a system's security and access control policy. HPol enables the formal representation of permissions in the form of tuples: Who, What, Which. It also enables the formal response to the generic access control question of *Who* (the subject) can perform *What* action on *Which* resource or object. HPol represents each *(Subject, Action, Object)* tuple with a node within a directed acyclic graph (DAG). Policy links between nodes connect subject, action, object nodes within the DAG to indicate what policies are allowed. In an HPol formal model, if a policy can be traced from the Start node to the End node then the nodes in the policy path describe the allowed access. In implementing a Mandatory Access Control (MAC) scheme, permissions not explicitly allowed are disallowed.

A primary advantage of HPol, in contrast with all other known access control models, is that the model's structure organizes subjects, actions, and objects using graph-based hierarchies. This enables the model to formalize these organizational hierarchies and enables the formal definition of abstraction and other formal or algebraic operations with system security policies such as policy concatenation and merging. An article describing the HPol formal model in a comprehensive manner, and all its potential uses, is currently under preparation.

Figure 4.3 displays the HPol model for a simplified file-system access control policy example. In the model the user Alice is allowed **read** permissions (or actions), by policy 1001, to her home directory **/home/alice** and **write** permissions, by policy 1002, also to her home directory **/home/alice**.

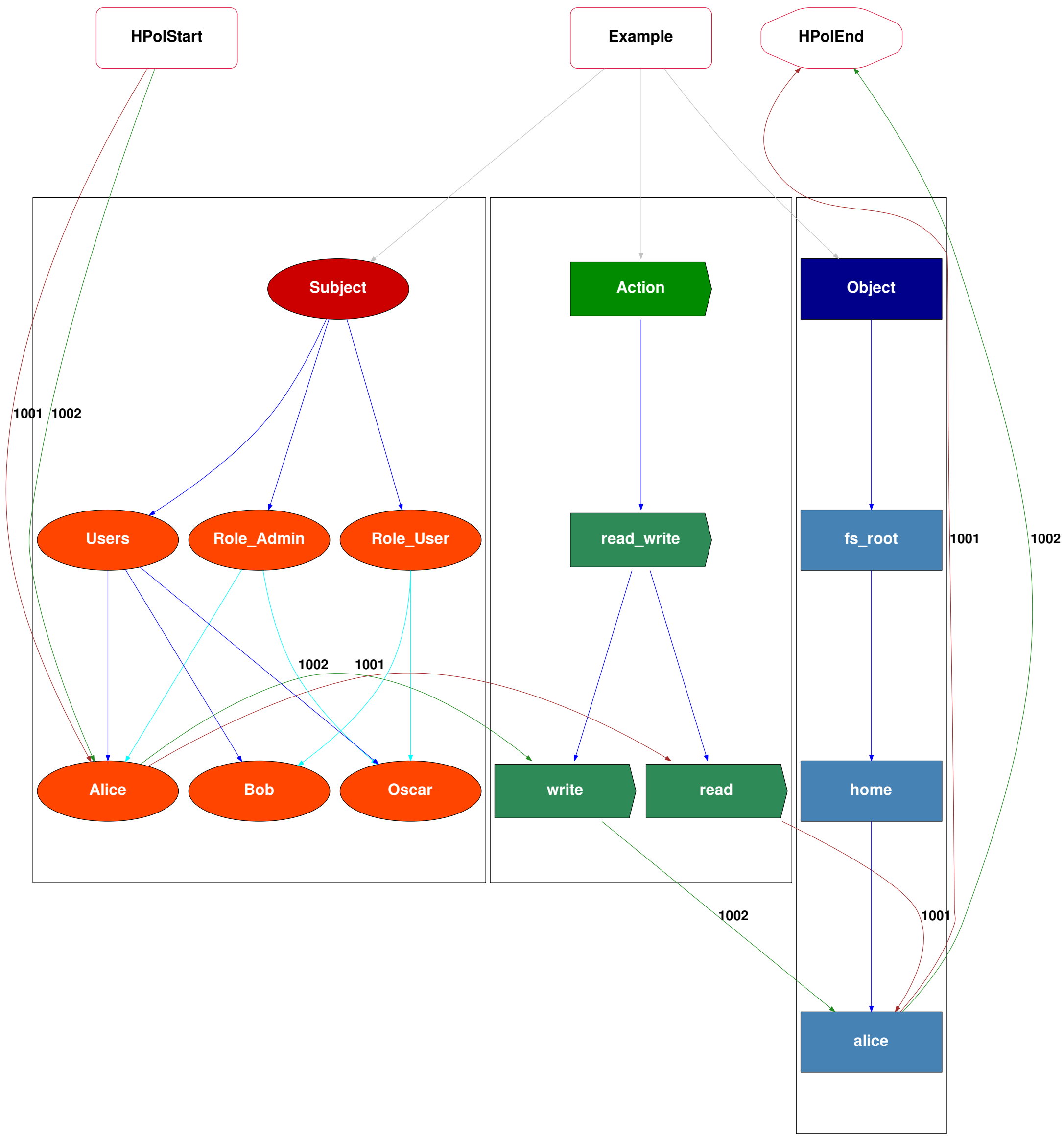


Figure 4.3: HPol Model: A simple example of the HPol formal model for a file system containing three different users, two permission types (file system read and write), and a single home directory.

4.3 The HERMES Specification for a Web Application

The High-Level Easily Reconfigurable Machine Environment Specification (HERMES) and its associated tool-set, was developed as a high-level security policy description language that enables system administrators to write web browser security policies that could then be implemented across the organization. HERMES enables the specification of organizational and domain hierarchies and the specification of policies at any desired level of abstraction. Originally, HERMES allows a policy designer to specify a given action or prohibition that applies to all browsers within an organization or to a single browser, for a single device, for a single user.

HERMES has evolved from a browser only specification to systemwide specification that enables the automatic generation of policy configurations based on a high-level policy specification. HERMES was designed with a goal of being written and read by humans rather than computers; therefore, HERMES is a text-based language capable of specifying security policies for many different system wide configurations including security configurations, including all browsers in any platform in any type of organization.

A HERMES policy specification is written in entity blocks. Entities have two components: head and body. An Entity Head corresponds to entity type and an identifier. An Entity Head contains only node, policy, or link. The Entity Body defines a set of fields or attributes and the order of these does not change the semantics. HERMES was designed using a context-free [60], definite-clause [61] , and block-like grammar format [62]. The BNF (Backus-Naur Form) specification of HERMES is provided in Figure 4.4. Figure 4.5 provides the syntactic usage.

HERMES BNF GRAMMAR

```
⟨File⟩                          ::= ⟨EntityList⟩
                                 |  EOF

⟨EntityList⟩                    ::= ⟨Entity⟩ ⟨EntityList⟩
                                 |  ⟨Entity⟩

⟨Entity⟩                        ::= ⟨EntityHead⟩ ⟨EntityBody⟩

⟨EntityHead⟩                    ::= ⟨EntityType⟩ : ⟨EntityTypeIdentifier⟩

⟨EntityType⟩                    ::= Node
                                 |  Policy
                                 |  Link

⟨EntityTypeIdentifier⟩          ::= ⟨Symbol⟩

⟨EntityBody⟩                    ::= { ⟨EntityBlockList⟩ } .

⟨EntityBlockList⟩               ::= ⟨EntityBlock⟩ ⟨EntityBlockList⟩
                                 |  ⟨EntityBlock⟩

⟨ EntityBlock⟩                  ::= ⟨BlockMemberIdentifierName⟩ : ⟨BlockMemberIdentifierValue⟩ ;

⟨BlockMemberIdentifierName⟩ ::= ⟨IdentifierSymbol⟩

⟨BlockMemberIdentifierValue⟩ ::= ⟨IdentifierSymbol⟩
                                 |  ⟨IdentifierString⟩
                                 |  ⟨IdentifierList⟩
                                 |  ⟨IdentifierDictionary⟩

⟨IdentifierSymbol⟩              ::= ⟨IdentifierChar⟩ ⟨IdentifierSymbol⟩

⟨IdentifierChar⟩                ::= A - Z
                                 |  a - z
                                 |  0 - 9
                                 |  .

⟨IdentifierString⟩              ::= ⟨String⟩

⟨IdentifierList⟩                ::= [ ⟨ItemList⟩ ]
                                 |  [ None ]

⟨ItemList⟩                      ::= ⟨item⟩ , ⟨ItemList⟩
                                 |  ⟨item⟩

⟨item⟩                          ::= ⟨Symbol⟩
                                 |  ⟨String⟩

⟨IdentifierDictionary⟩          ::= { ⟨KeyValuePairList⟩ }

⟨KeyValuePairList⟩              ::= ⟨Key⟩ : ⟨Value⟩, ⟨KeyValuePairList⟩
                                 |  ⟨Key⟩ : ⟨Value⟩

⟨Key⟩                           ::= ⟨Symbol⟩
                                 |  ⟨String⟩

⟨Value⟩                         ::= ⟨Symbol⟩
                                 |  ⟨String⟩

⟨Symbol⟩                        ::= SymbolLiteral

⟨String⟩                        ::= 'StringLiteral'
                                 |  "StringLiteral"
```

Figure 4.4: HERMES: The formal specification of the HERMES grammar

HERMES BNF SYNTACTIC USAGE

```
Node: TypeIdentifier
{
    SymbolLiteral: SymbolLiteral ;
    SymbolLiteral: 'StringLiteral' ;
    SymbolLiteral: "StringLiteral" ;
    SymbolLiteral: [ IdentifierList ] ;
    SymbolLiteral: { IdentifierDictionary } ;
} .

Policy: TypeIdentifier
{
    SymbolLiteral: SymbolLiteral ;
    SymbolLiteral: 'StringLiteral' ;
    SymbolLiteral: "StringLiteral" ;
    SymbolLiteral: [ IdentifierList ] ;
    SymbolLiteral: { IdentifierDictionary } ;
} .

IdentifierList = [None] or [ item] or [ item1, item2, item3, ..., itemN ]

item = SymbolLiteral or 'StringLiteral' or "StringLiteral"

IdentifierDictionary = { Key : Value } or
                       { Key1 : Value1, Key2 : Value2, ..., KeyN : ValueN }

Key = SymbolLiteral or 'StringLiteral' or "StringLiteral"

Value = SymbolLiteral or 'StringLiteral' or "StringLiteral"
```

Figure 4.5: HERMES Usage: The formal usage specification for the HERMES grammar as previously outlined.

4.4 Manually Building a Formal Web Application Security Model

In this section the HPol security model is applied to the Mutillidae Web application. In order to create a security model, the Mutillidae Web application was reverse engineered. In reverse engineering the Web application the subjects, actions, and objects were identified. In this instance the subjects are identified as the individual Web pages, and either the Mutillidae database user (dbRoot) or the Mutillidae Web application Apache user (www-data). The actions are identified as the permissions allowed on the Mutillidae file-system or the SQL query allowed on the DBMS. The objects are identified as the individual Web pages of the Mutillidae Web application and the Mutillidae MySQL database tables. Figure 4.6 illustrates the manual process for building the formal web application security model. The web application was reversed engineered for the filesystem and database queries.

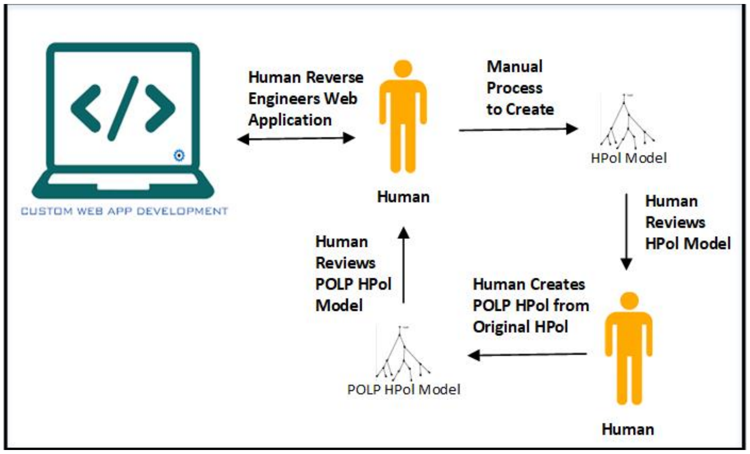


Figure 4.6: Manual HPol: The manual process of building the security model.

4.5 Manually Building a Formal Web Application Security Model: Filesystem

The Mutillidae Web application Apache was installed on a Linux Ubuntu server. The location of the Apache configuration files is */etc/apache2*. Within these configuration files, Apache describes a single user known as *www-data*. The actual location of any files within the Web server are located in */var/www/html*. By default *www-data* does not have the proper permissions to take action on the directories in */var/www/html*. For ease of access, the Mutillidae installation directions recommend changing the permission on */var/www/html* to be readable, writable, and executable for all users and all groups (*rwx*) [14].

The Mutillidae file-system resides in */var/www/html/mutillidae* within the host Linux system. Within the Mutillidae directory is a set of Web pages and other directories. The Web pages with the Mutillidea directory are *index.php*, *login.php*, *register.php*, *addToBlog.php*, *uploadFile.php*, *captureData.php*, *viewBlog.php*, and *userInfo.php*. The subdirectories within the Mutillidae directory are *classes*, *data*, and *includes*. The *classes* subdirectory contains the PHP code connectivity to the Mutillidae MySQL database. The *includes* directory contains the common PHP files that all Web pages in the Mutillidae Web application use.

Figure 4.7 illustrates the policies for the *uploadFile.php* Web page. The Web page *uploadFile.php* has filesystem access via the Mutillidae Apache user *www-data*. The Mutillidae

Apache user *www-data* can perform the action *rwx* on the object Mutillidae file-system directory *html*. Although *Mutillidae* is a subdirectory of */var/www/html* the permissions are applied to */var/www/html/* and all of the subdirectories below the *html* directory.

Figure 4.7 illustrates that the permissions on the Mutillidae directory of */var/www/html/-mutillidae*, including the Web pages and the subdirectories, are read, write, and execute (*rwx*) for the *www-data* user. This is problematic because the *classes* subdirectory contains the PHP file *RemoteFileHandler.php*. This PHP file contains the code and permissions to upload a file. A malicious user could use the security misconfiguration of *RemoteFileHandler.php* to upload a malicious file that could corrupt the Linux server.

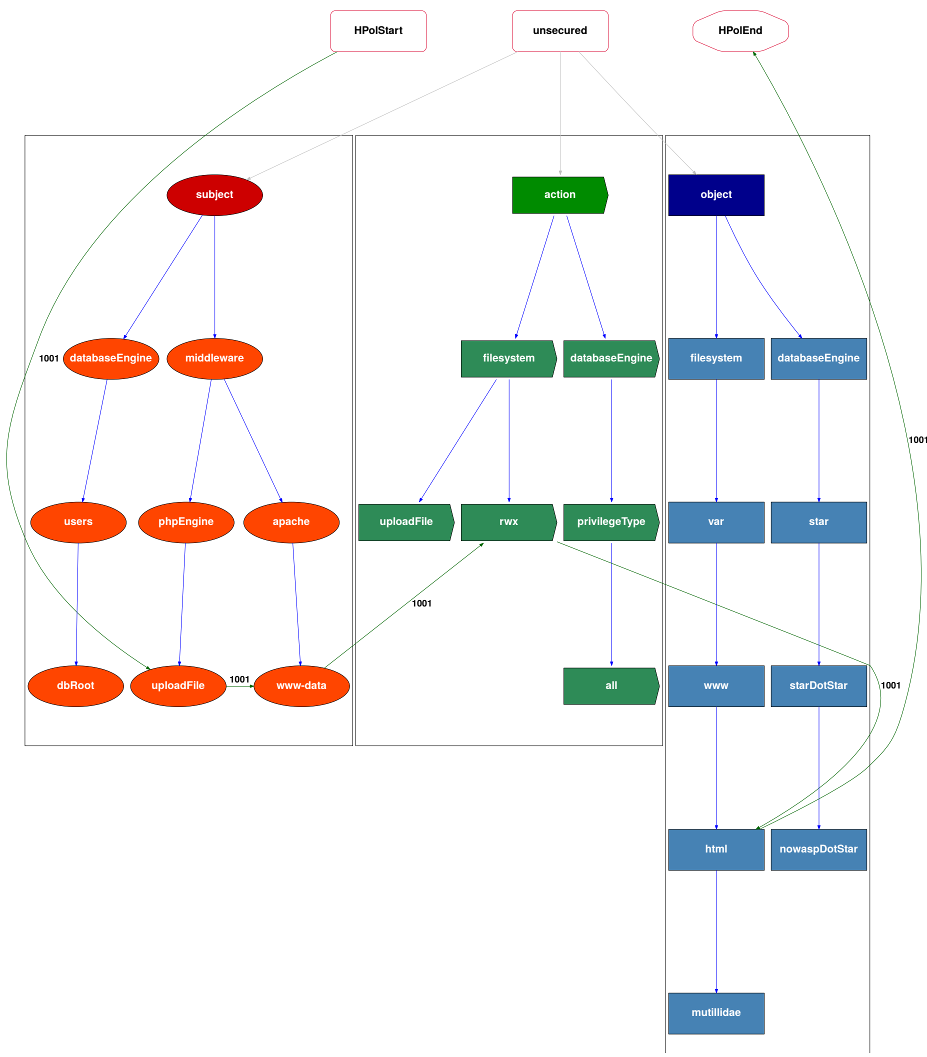


Figure 4.7: HPol model of the Mutillidae File-system: This figure illustrates the insecure file-system privileges.

4.6 Manually Building a Formal Web Application Security Model: DBMS

For the Mutillidae Web application, the database management system (DBMS) is *MySQL*. The set of available MySQL commands are, `show databases`, `select`, `alter`, `drop`, `create`, `delete`, `insert`, `shutdown`, `process`, `execute`, and `update`. The Mutillidae Web application contains a class *classes/MySQLHandler.php* with a single privileged user named root. This root user has full access to all commands in the DBMS. Figure 4.8 illustrates the creation of the single privileged root user in the file *MySQLHandler.php*.

A Mutillidae Web page executes a query into the database via the PHP file *classes/SQLQueryHandler.php*. The Web page calls *SQLQueryHandler.php* which constructs the MySQL query. Once the query is constructed the *root* user issues the query. The action is the SQL query. The HPol Mutillidae object is the database table, database procedure, or the database function referenced by the SQL query.

For example an end-user attempting to login to their Mutillidae account would execute the page *login.php*. This login page prompts the user for their username and password. Once the username and password are entered and the submit button is pressed the function *authenticateAccount* is called in the file *SQLQueryHandler.php*. This PHP file builds the SQL query. Figure 4.9 illustrates the constructed SQL query after the user inputs their username and password on the *login.php* page.

```
/* ----------------------------------------------
 * DATABASE USER NAME
 * ----------------------------------------------
 * This is the user name of the account on the database
 * which OWASP Mutillidae II will use to connect. If this is set
 * incorrectly, OWASP Mutillidae II is not going to be able
 * to connect to the database.
 * */
static public $mMySQLDatabaseUsername = "root";
```

Figure 4.8: Web Application Root: A code excerpt illustrating the connection from the middleware to the DBMS using root level privileges. (Mutillidae:2.6.42 GPL mutillidae/classes/MySQLHandler.php)

```php
public function authenticateAccount($pUsername, $pPassword)
{

        $lQueryString =
                "SELECT username ".
                "FROM accounts ".
                "WHERE username='".$pUsername."' ".
                "AND password='".$pPassword."';";

        $lQueryResult = $this->mMySQLHandler->executeQuery($lQueryString);

        if (isset($lQueryResult->num_rows)){
                return ($lQueryResult->num_rows > 0);
        }else{
                return FALSE;
        }// end if

}//end public function
```

Figure 4.9: Web Application Query: This figure illustrates the the query string in SQLQueryHandler.php for a login attempt. (Mutillidae:2.6.42 GPL mutillidae/classes/SQLQueryHandler.php)

Figure 4.10 illustrates the DBMS access permissions of the Mutillidae web application. In this figure it can be observed that *root* can perform any SQL command on any database, table, function or procedure within the MySQL DBMS. In the HPol Mutillidae security model the `dbRoot` user is the subject. The actions are the possible SQL queries such as `select`, `update`, `insert`. The objects are a hierarchical structure under the MySQL DBMS. At the top of hierarchy is the regular expression `star` which represents zero or more databases or database tables. Below `star` is `starDotStar`. Below `starDotStar` is the `mysqlDotStar`. Below `mysqlDotStar` are the database *tables*, *functions* and *procedures* for that database.

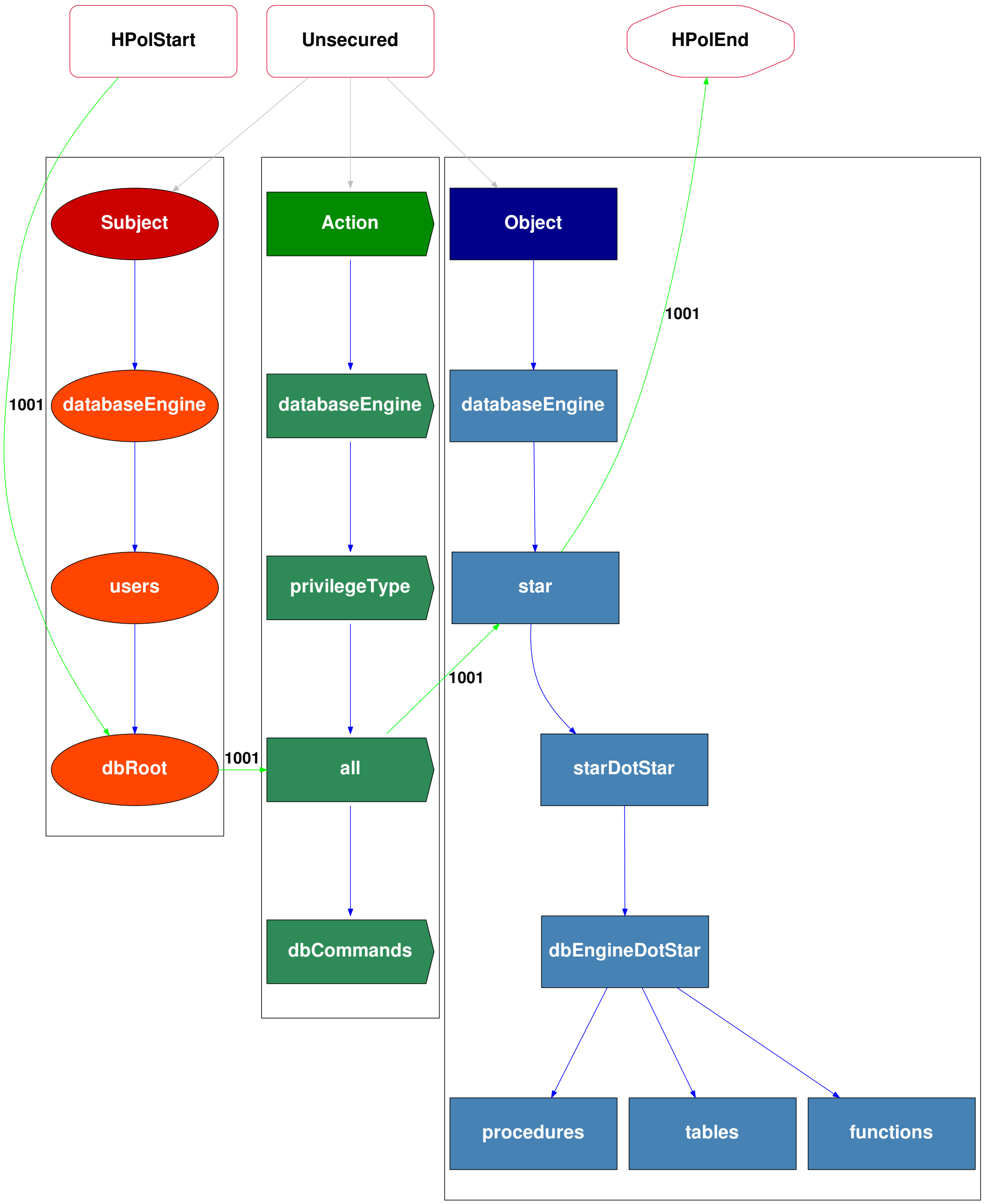


Figure 4.10: HPol Model of the Current Mutillidae Web Application: Illustrating Non-Least Privilege (insecure) Mutillidae DBMS (MySQL) Access Permissions.

4.7 Applied Case Study: Mutillidae - Manually Applying HPol

In Section 4.4 the process of reverse engineering and applying the HPol formal security model is described. In this section, a step by step process using the case study of the Mutillidae web application is described. The step by step process is applied to the filesystem, and then the database management system.

OWASP Mutillidae II is a deliberately vulnerable Web application. Mutillidae II may be used by developers to learn secure Web coding practices. It uses a Web Server, such as Apache, plus PHP for middleware and a DBMS back-end, such as MySQL or MariaDB. Mutillidae II may be installed on Linux, Windows, or MacOS using a LAMP, WAMP, or XAMMP application stack [14]. Mutillidae II was used as a case study for this research for the following reasons: (1) Richness of available instructional code examples; (2) Ability to change the security-level and implement and test different vulnerability mitigation strategies; (3) Availability of complete source code and flexible license (GPL3); and (4) Uses PHP, the current target language of this research due to its widespread use in Web applications.

Mutillidae II was developed to teach secure Web application development and it does not implement a least-privilege secure design approach. Rather it uses the widespread practice of highest privilege approach of granting administrative-level permissions to the middleware on the DBMS. In other words, Mutillidae, similarly to most other learning-focused Web applications, focuses its instructional approach on bettering the practice of secure coding but not on bettering the practice of secure application design. We believe the latter to be as important or more than the former, however, likely harder to master and implement.

Manually Applying HPol to the Filesystem

Step 1: Identify the file-system permissions required to move from non-least privilege to least privilege. The appropriate file-system permissions for each Web page in the Mutillidae Web application are displayed in Table 4.1.

Subject: Linux User	Object: Type	Object: Name	Actions: Current Non-POLP	Actions: New POLP
www-data	Page	index.php	read, write, execute	read, execute
www-data	Page	login.php	read, write, execute	read, execute
www-data	Page	register.php	read, write, execute	read, execute
www-data	Page	addToBlog.php	read, write, execute	read, execute
www-data	Page	captureData.php	read, write, execute	read, execute
www-data	Page	viewBlog.php	read, write, execute	read, execute
www-data	Page	userInfo.php	read, write, execute	read, execute
www-data	Page	fileUpload	read, write, execute	read, execute
www-data	Directory	classes, includes	read, write, execute	read, execute
www-data	Directory	images	read, write, execute	read

Table 4.1: Current (Non Least Privilege) and New (Least Privilege or POLP) Linux file-system permissions for each PHP web-page file and file storage directories in the Mutillidate Web application.

For example the *index.php* page displays information about the Mutillidae Web application as well as it contains links to open other Web pages. The column labeled `Actions: Current Non-POLP`, in Table 4.1 shows that the current file-system permissions are set to read, write, and execute. The principle of least privilege states the appropriate file-system permissions for the *index.php* page should be read only, to display the contents of the page, and execute only for opening the links to other pages. This is shown in the column labeled `Actions:New POLP`.

Step 2: Modify the Web application file-system permissions for the file-system directories and PHP files. The appropriate HPol security model for the POLP requires the permissions of `rwx` be broken into read and execute permissions (`rx`) and write only permissions under the node `rwx`. The node `rx` (read and execute) is further broken in to read only and execute only permissions. The permissions on */var/www/html* are changed to read and execute, with no write permissions.

Furthermore, the permissions must be changed on other files that are not Web pages and directories, using the command *sudo chown -R $USER:www-data /var/www* and the command *sudo chmod -R 640 /var/www*. The Mutillidae Web application has subdirectories named *classes*, *data*, and *includes*. Recall the *classes* subdirectory contains the

PHP files for handling file uploads, building query strings, and database connectivity. The required permissions on the *classes* subdirectory are required to be read only and execute only (**rx**). To move the *index.php* page from non-least privilege to least privilege the file-system permissions were changed to be read and execute (*rx*) for the Linux system user and Linux system group only. Figure 4.11 illustrates how the file-system permissions have been corrected for the *index.php* page.

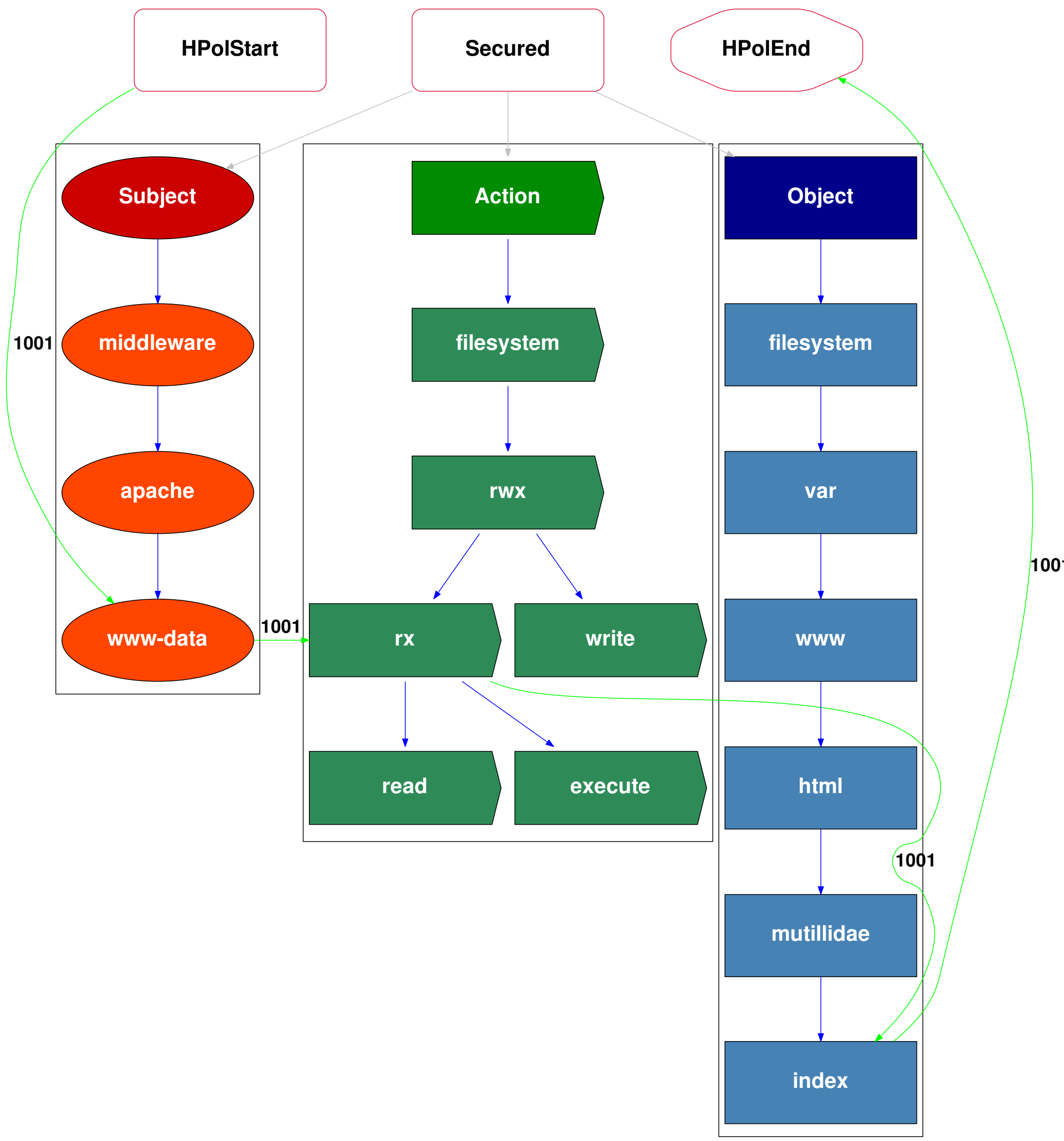


Figure 4.11: Least Privilege HPol Model: This figure illustrates the least privilege model of the web page index.php and the interaction with the web server.

Step 3: Modify the Linux file-system permissions for the Apache user. Recall the Apache Web server runs under a Linux system user named *www-data*. This *www-data* Linux system user resides outside the Mutillidae Web application, this user currently contains read, write and execute permissions. The *www-data* user should have read and execute permissions only. The permissions on the *www-data* user were modified by the command **chmod g+s**. This command allows all new files and subdirectories created by the *www-data* user to inherit the group ID of the directory. Since the group ID permissions were changed for all directories and subdirectories in Step 2, any changes to the file-system by the *www-data* user by default are read and execute (`rx`). This is important for file uploads. Any file that is uploaded is now uploaded without write permissions, which prevents malicious code that is uploaded from making changes to the Linux file-system. Table 4.1 illustrates the new permissions for the *fielUpload* page are now set to read and execute.

Manually Applying HPol to the Database

Step 1: Identify the DBMS and Mutillidae database permissions required to move from non-least privilege to least privilege. The appropriate database permissions, including least privilege database users, and least privilege database commands for each Web page in the Mutillidae Web application are displayed in Table 4.2.

For example, the *login.php* page, from Figure 4.12, prompts the user to enter their username and password. Recall that once the username and password are entered and the submit button is pressed, the function *authenticateAccount* is called. The *authenticateAccount* function creates a query using only the `select` statement. Since only the `select` statement is issued the HPol subject specifies the *loginSelectAccounts* only has permissions to issue the `select` database command. Figure 4.14 illustrates the updated HPol model. The new user loginSelectAccounts has been created and applied to the page *login.php*.

Step 2: Remove the root user from the DBMS connection. The single privileged root user used to connect the DBMS and shown in Figure 4.8 is abandoned. A new set of least privilege users are created with the permissions needed for each individual Web page.

Subject: Page Name	Subject: POLP User	Action: POLP Permissions	Object: DBMS Table
index.php	none	none	none
login.php	loginSelectAccounts	select	accounts
userInfo.php	userInfoSelectAccounts	select	accounts
viewBlog.php	viewBlogSelectAccBlogs	select	accounts blogs_table
addToBlog.php	addToBlogSelectAccInsBlogs	select insert	accounts blogs_table
register.php	registerSelectInsertAccounts	select insert	accounts accounts
captureData.php	captureDataInsertCaptureData	insert	capture_data

Table 4.2: New POLP permissions for each web page and the corresponding new user and restricted permissions on the DBMS.

Continuing the example of *index.php* the `mysql.users` table within the database was updated to include the new user. The new user was created by using the page name `login`, the SQL command `select`, and the database table `accounts`. Figure 4.13 illustrates the new least privilege user. Figure 4.13 illustrates the updated HPol model where the new user *loginSelectAccounts* has been created and applied to the page *login.php*.

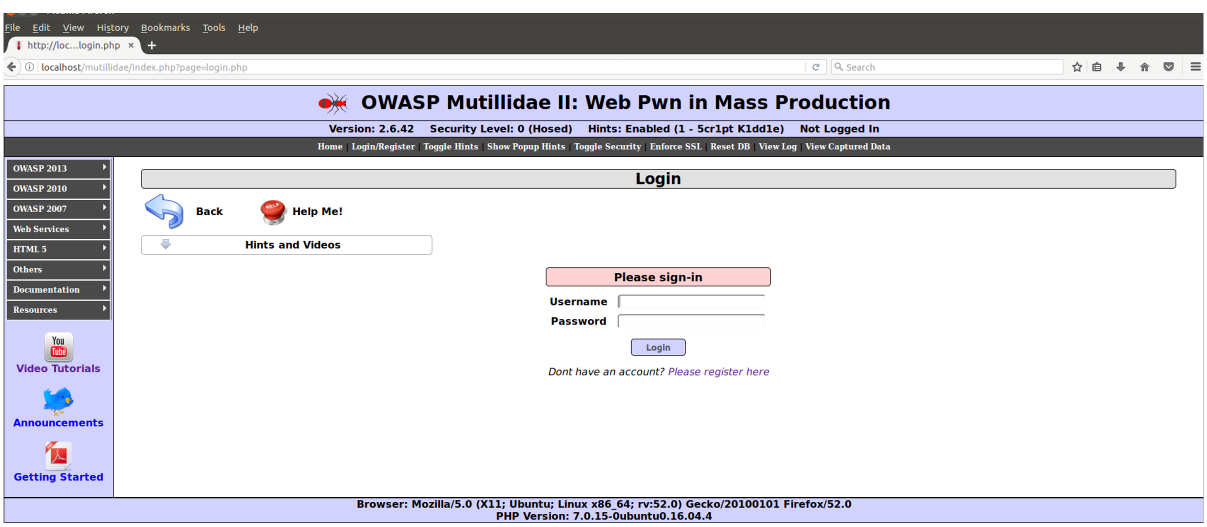
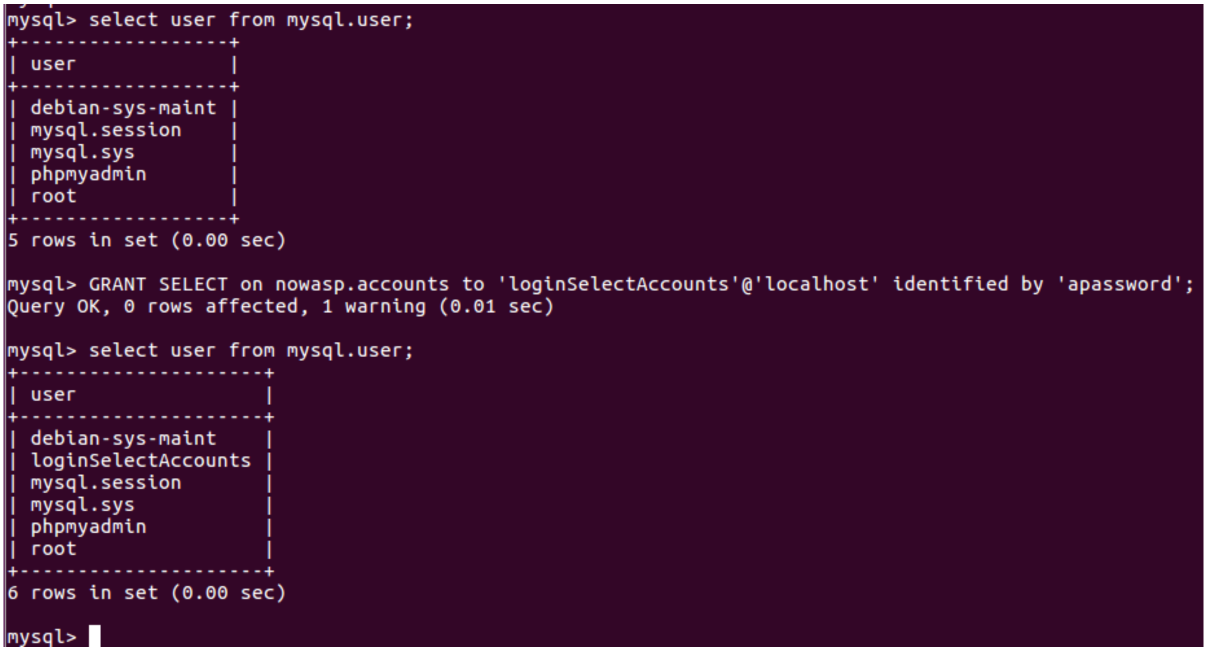


Figure 4.12: Mutillidae Login: This figure illustrates the Mutillidae login screen as displayed from the Mutillidae Web application.



```
mysql> select user from mysql.user;
+------------------+
| user             |
+------------------+
| debian-sys-maint |
| mysql.session    |
| mysql.sys        |
| phpmyadmin       |
| root             |
+------------------+
5 rows in set (0.00 sec)

mysql> GRANT SELECT on nowasp.accounts to 'loginSelectAccounts'@'localhost' identified by 'apassword';
Query OK, 0 rows affected, 1 warning (0.01 sec)

mysql> select user from mysql.user;
+---------------------+
| user                |
+---------------------+
| debian-sys-maint    |
| loginSelectAccounts |
| mysql.session       |
| mysql.sys           |
| phpmyadmin          |
| root                |
+---------------------+
6 rows in set (0.00 sec)

mysql>
```

Figure 4.13: Principle Of Least Privilege Database: The POLP applied to the nowasp MySQL database. This figure illustrates the MySQL commands to grant the user limited privileges on the accounts table within the database.

Step 3: The modification of the database commands. The appropriate HPol security model for the POLP requires that the action `dbCommands` be broken into individual database commands such as `select_insert`. Subsequently, the newly created action node `select_insert` is further broken into `select` only and `insert` only. This occurs for all database commands allowed in the MySQL database engine (DBMS).

As an example, returning to the *login.php* page the individual required database command is `select`. In Step 2 the specific least privilege subject *loginSelectAccounts* was created. In Step 3 the least privilege action `select` is called by the least privilege subject *loginSelectAccounts*.

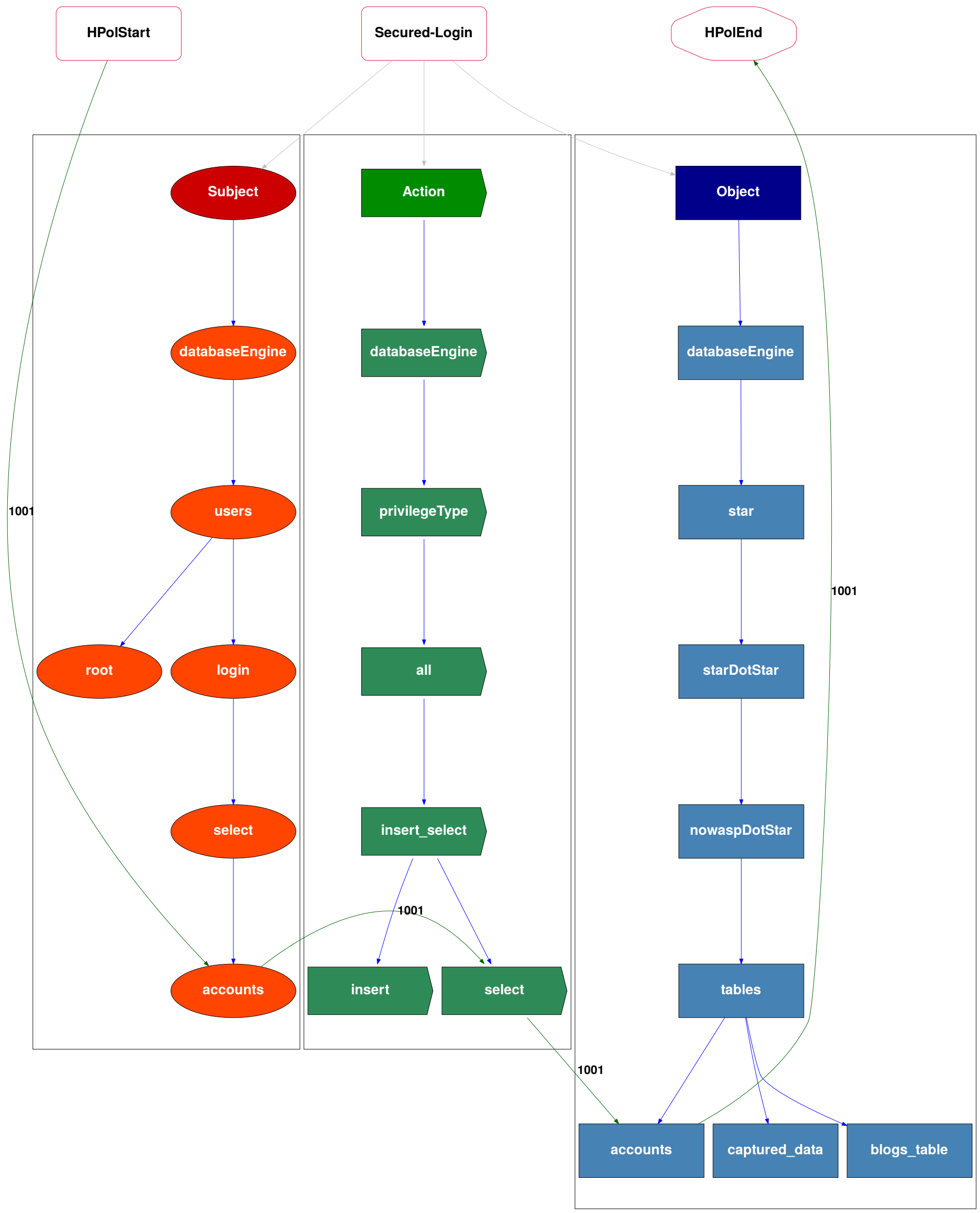


Figure 4.14: Principle Of Least Privilege Web Page: This figure illustrates the POLP applied to login.php. The POLP illustrates the subject and action only has permissions for the SELECT statement and the object is only for the accounts table.

Step 4: Modification of the the Mutillidae database tables. The appropriate HPol security model for the POLP requires that the object *star* be restricted to the individual database tables in the Mutillidae Web application.

For example, the *login.php* page in Step 2 requires a *loginSelectAccounts* as the HPol subject. In Step 3 the *login.php* page requires only the database command `select` as the HPol action. In Step 4 the HPol object for the *login.php* page, requires access only to the *accounts* table.

Step 5: Systematically apply the POLP to each subject web page that requires database access. This requires understanding the SQL commands required for each page, and understanding each database table that the page accesses. Similar to Step 4, a new SQL user is created representing the page, the SQL command and the database table. For example, the *add-to-your-blog* requires two new SQL users. First, *add-to-your-blog* queries the `accounts` table for the blog user. Once the user is authenticated, the *add-to-your-blog* page inserts the blog comment into the `blogs_table`. This set of queries can be summarized as *add-to-your-blog* page, executes a `select` command on the `accounts` table. Once the user is authenticated the *add-to-your-blog* page, executes an `insert` command on the `accounts` table.

Figure 4.15 illustrates the principle of least privilege applied to the *add-to-your-blog.php* page. The new users are illustrated with the path `Subject/databaseEngine/users/addToYourBlog`. The HPol security model defines two policies for this add-to-your-blog Mutillidae Web page. Policy 1001 illustrates the subject *addToYourBlogSelectAccounts* can only perform the action of executing the database `select` command for the database table *accounts*. After Policy 1001 completes, then Policy 1002 allows the subject *addToYourBlogInsertBlogsTable* to only perform the action of executing the database `insert` command for the database table *blogs_table*.

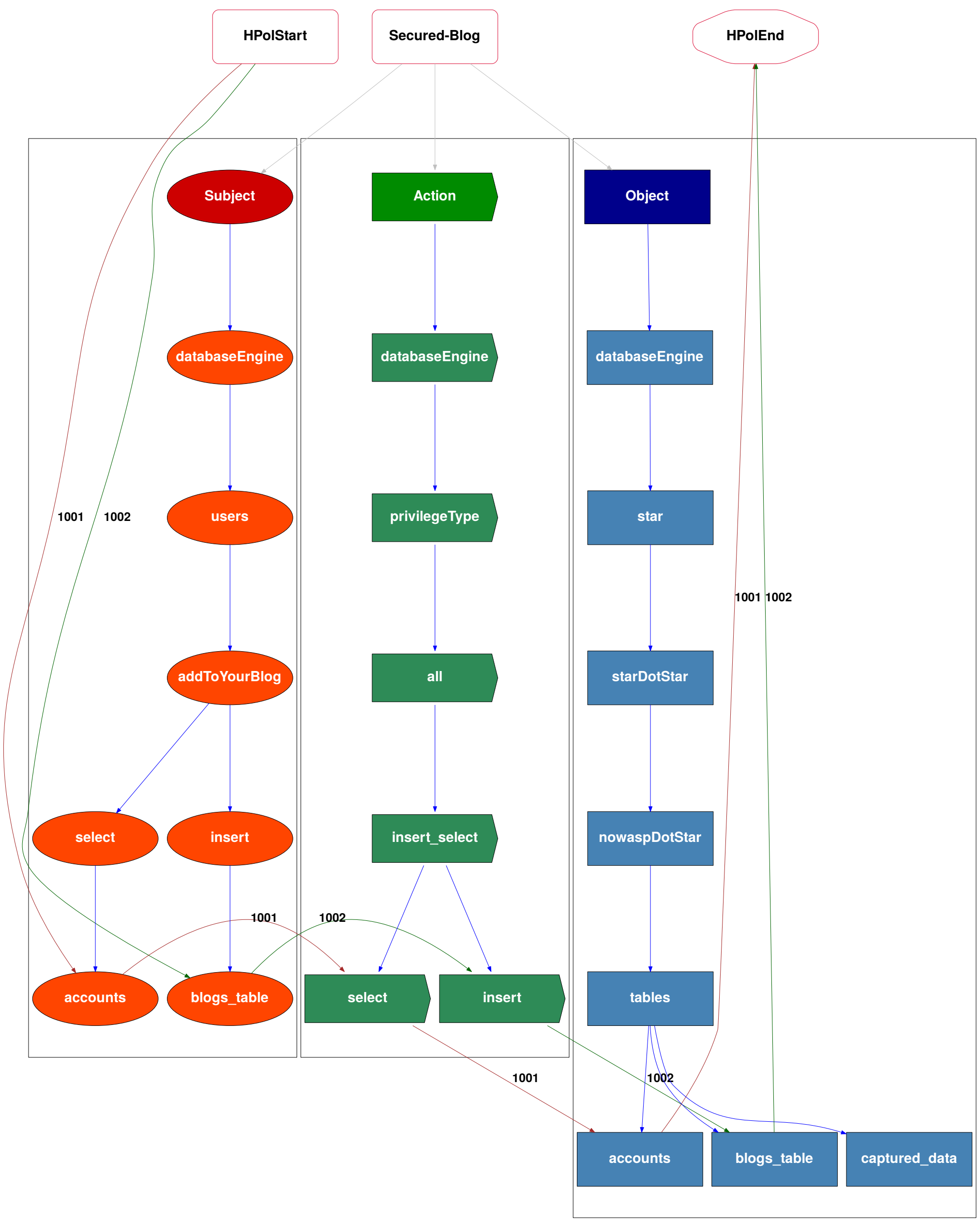


Figure 4.15: Principle Of Least Privilege Web Page - addToBlog: This figure illustrates the updated PHP code enforcing the Principle of Least Privilege for the page addToBlog.

Chapter 5: Enhancements to HPol and HERMES for Increased Web Application Security

This chapter explains the enhancements to HPol and HERMES developed for this dissertation. These enhancements were required for the semi-automated approach presented in this dissertation. This chapter outlines **Contribution 2: Formal web application security policy modeling** explained in Section 5.1. and **Contribution 3: Formal High-Level Easily Reconfigurable Specification** explained in Section 5.2. Both contributions are illustrated in Figure 5.1. The practical application of HPol and HERMES is demonstrated in Section 5.3.

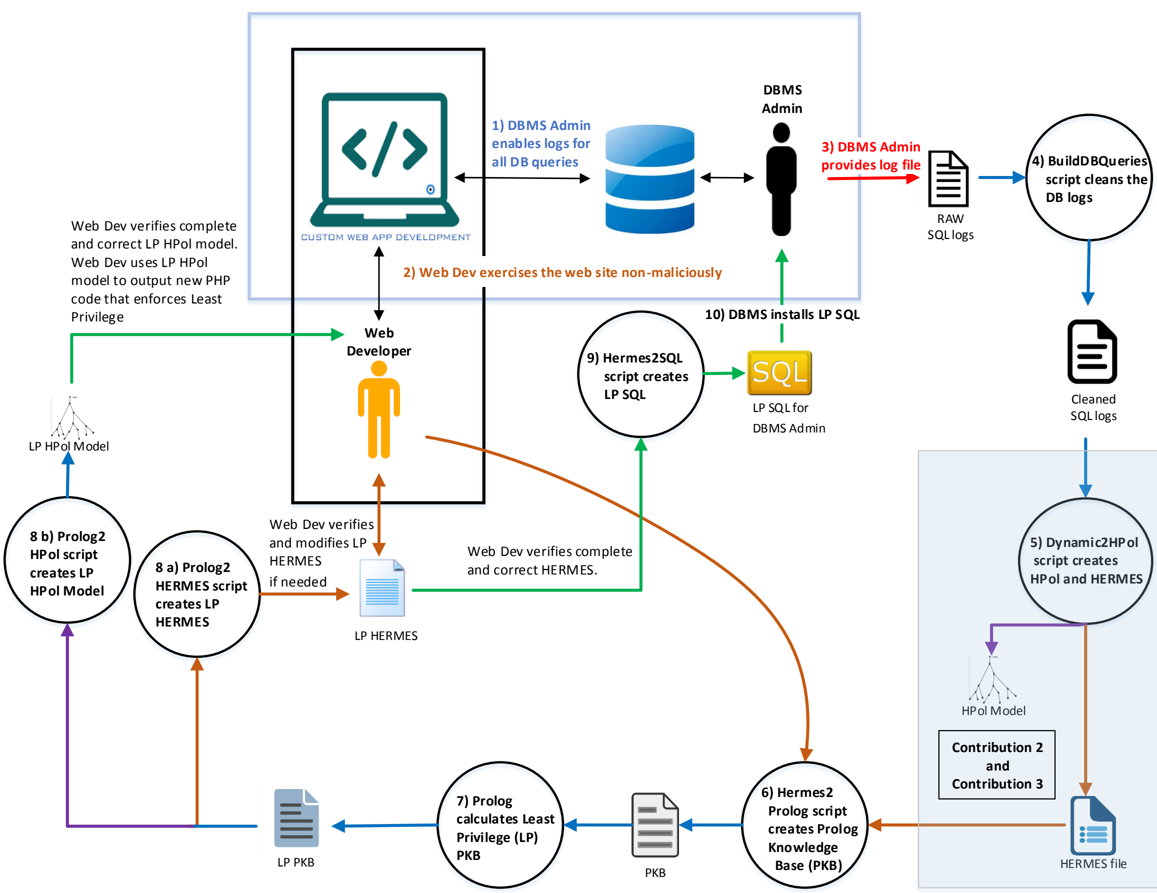


Figure 5.1: Contribution 2: Formal web application security policy modeling. Contribution 3: Formal HERMES Specification.

5.1 Enhancing the HPol Formal Security Model

The purpose of The Hierarchical Policy (HPol) formal model, and its associated tool-set, was to enable a formal representation of a system's security and access control policy. In order to model web application security the following enhancements were made to HPol:

Step 1: The concepts of domains was added to HPol. Domain names are used to identify the particular website. The domain was added to separate one web application from another web application with the same name. An example would one web application with the URL of http://www.somewebapp.com/ versus a completely different web application with the URL http://www.somewebapp.net/.

Step 2: A hierarchical namespace was added to HPol. Similar to the domain the purpose of the namespace was to group like structures (web pages, networks, etc.) to avoid name collisions for multiple identifiers that might share the same name within a domain.

Step 3: The Node class was originally embedded inside the HPol Python file. This class was extracted into its own stand alone class. This now allows for an HPol object to be created without having to create a node. Since HERMES can be tightly coupled (not required) with HPol, having the first order predicate Node separated enabled ease of use for both HPol and HERMES.

Step 4: Parts of the Policy class were originally embedded inside the HPol Python file. The parts necessary for this dissertation were extracted into a stand alone class file. Similar to the Node class, Policy is a first order predicated and will eventually become a complete stand alone class.

Step 5: A rudimentary Link class was added to HPol. Similar to the Policy class and the Node class, the Link class is also a first order object. The Link class will also aid in previous research concerning Cisco router policy [63] when the class is complete.

Step 6: A fully qualified name (FQN) was added to HPol. By default, a node's path goes from the start of the HPol DAG to the terminal node. That path, coupled with the namespace and the domain define a unique FQN.

Step 7: User numbered policies were added to HPol. The original default behavior was to number policies starting from 1000. In order to convert from a non-least privilege security model to a least privilege security model the ability to add a policy with a predefined number was required. The HPol consistency checker ensures there are no duplicate policy numbers.

Step 8: The ability to create a HERMES file was added to HPol. In order to be a semi-automated approach the ability to create a HERMES file was required.

Step 9: Minor code cleanup and refactoring was done to aid in the research in this dissertation.

HPol was an excellent stand alone tool set that was being utilized in many different research activities. These enhancements were necessary to create the semi-automated approach of this dissertation to prevent cyber attacks on web applications.

5.2 Enhancing and Formally Defining the HERMES Language

Recall that the purpose of the High-Level Easily Reconfigurable Machine Environment Specification (HERMES) language, and its associated tool-set, was originally for granular browser configuration. The HERMES specification was originally defined by Jillepalli and Conte de Leon [64]. In order to be able to move from non-least privilege to least privilege the following changes to HERMES were made:

Step 1: A formalized Entity Head was added to HERMES. HERMES contained an Entity Head; however, the values could be anything. By formalizing the values as Node, Policy, or Link, then these values can be coupled with HPol to provided dedicated first order predicates.

Step 2: A formalized Entity Body was added to HERMES. Although HERMES contained an Entity Body, that body could also be anything. By formalizing the body to be only symbols, stings, lists, or dictionaries allowed for ease of using HERMES to create a Prolog Knowledge Base (PKB).

Step 3: A formalized data structure similar to the HPol data structure was created for HERMES. This data structure allows for a HERMES file to be written in plain text by anyone; however, the HERMES data structure objects can be created from the text file and used for various purposes for this dissertation.

Step 4: The fully qualified name (FQN) was also added to HERMES. This name, similar to HPol FQN, allows for the exact Nodes, Files, or Policies to be specified.

Step 5: Minor refactoring and additional specifications were added. Every HERMES Entity Body starts with a left curly bracket ({) and ends in a right curly bracket and a period (}.) Furthermore, each line in the body ends in a semicolon.

Similar to HPol, the enhancements to HERMES were required for the work in this dissertation and to allow for the security model to move from non-least privilege to least privilege. For reference, the original HERMES specification is shown in Figure 5.2 and a portion of the new HERMES specification is shown in Figure 5.3.

5.3 Applied Case Study: Mutillidae - Applying HPol and HERMES Enhancements

In Chapter 4, the process of creating the security model for the Mutillidae web application was manually completed by reverse engineering the web application. In order to semi-automate the process, the enhancements to HPol and HERMES had to be completed. This section details how the enhancements of HPol and HERMES were tested for the Mutillidae web application.

The original version of HERMES allowed for any value at the Entity Head. Originally the Mutillidae web application contained a domain, subdomain, node and policy. Figure 5.2 illustrates the original version of HERMES which allowed any Entity Head. This old configuration made it difficult to build a HERMES data structure as well as convert the HERMES file to Prolog and from Prolog to least privilege.

The fully qualified name (FQN) does not match the entity, and the body contained a list of the children for the entity. Creating a FQN, and restricting the Entity Head to Node, Policy, or Link, simplified the conversion to least privilege.

```
Domain: HPol
{
      FQN: mutillidae.HPol.unsecured;
      Description: "HPol Root Node";
      Path: "unsecured";
      Type: "HPolRoot";
      Children: [object, subject, action];
}

SubDomain: object
{
      FQN: mutillidae.HPol.unsecured.object;
      Description: "object";
      Path: "unsecured/object";
      Type: "object";
      Children: [database];
}

Node: database
{
      FQN: mutillidae.HPol.unsecured.object.database;
      Description: " database ";
      Path: "unsecured/object/database";
      Type: "object";
      Children: [var];
}

Policy: PID_1001
{
      Description: "HPol Policy";
      Status: ENABLED;
      Path: [HPolStart, object, database, HPolEnd];
}
```

Figure 5.2: Original HERMES: This figure illustrates the original version of HERMES before the formal specification as outlined in this research.

The original HPol was created manually after reverse engineering the web application. In the original HPol the web pages were modeled as objects. This was problematic in that the web pages were being executed, and during that execution the web pages were building the queries that were being executed by the *root* web application user. Creating the enhancements to HPol and removing the Node Python class, allowed for quickly and easily changing the web pages from objects to subjects.

These changes allowed for simple but efficient data structures that quickly built the dynamic HPol security model and the dynamic HERMES model. Once these models were built, the conversion to and from Prolog allowed for easy conversion to least privilege.

```
Node: star
{
    FQN: Namespace.Domain.Example.Objects.databaseEngine.star;
    Description: "star";
    Path: "Example/Objects/databaseEngine/star";
    Type: "object";
}.

Policy: PID_1001
{
    FQN: Namespace.Domain.PID_1001;
    Description: 'Policy';
    Status: ENABLED;
    RelativePath: [HPolStart, dbRoot, all, star, HPolEnd];
}.
```

Figure 5.3: New HERMES: This figure illustrates the new version of HERMES after the formal specification as outlined in this research.

In summary, this chapter explained the enhancements to both the HPol security model and the HERMES language. These changes allowed for a more robust dynamic solution for building the models and converting the models from non-least privilege to least privilege.

Chapter 6: Automating Learning the Least Privilege Policy for a Web Application

With a robust set of tools, the next step in the process is the inference of the exact non-least privilege security model for the web application. The process of automating the inference is explained in Section 6.1. The practical application of determining the non-least privilege is applied in two case studies. The inference of the Security Education (SEED) Labs is explained in Section 6.2 and the inference of the Mutillidae web application is explained in Section 6.3. Figure 6.1 illustrates **Contribution 4: The associated tools for automatically learning the database-level permissions needed to operate with least privilege** of this dissertation.

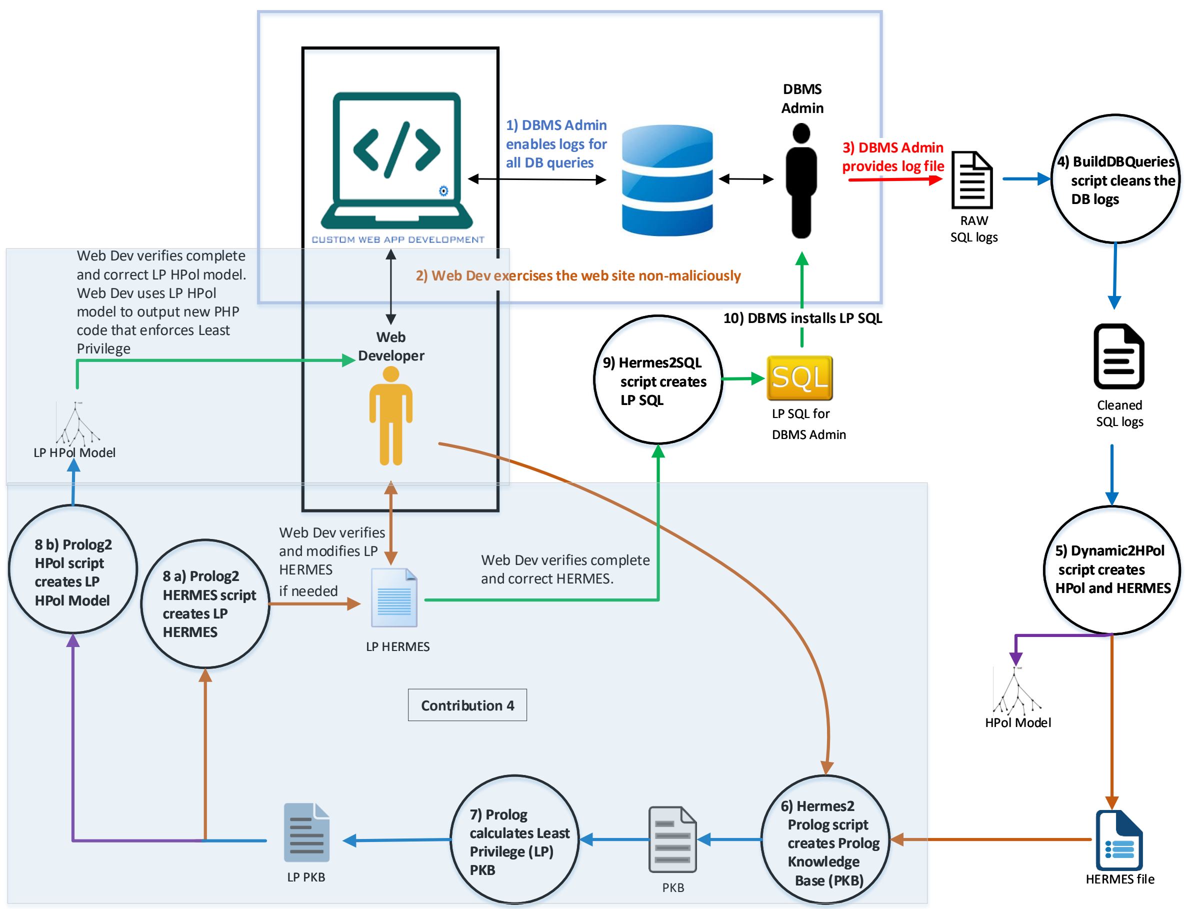


Figure 6.1: Contribution 4: The associated tools for automatically learning the database-level permissions needed to operate with least privilege as represented from the architectural overview of this dissertation.

6.1 Systematic Inference of DB Table and SQL Command Level Access Control

To infer the exact non-least privilege security model the general database log of the web application must be provided. These database logs contain all the database commands of the website, with the website exercised in a non-malicious manner.

Step 1: The DBMS administrator turns on the general logs with the following commands, in no specific order.

- `SET GLOBAL log_output = 'FILE';`
- `SET GLOBAL general_log = 'ON';`
- `SET GLOBAL general_log_file = '/var/lib/mysql/filename.log';`

After logging is enabled, the DMBS administrator notifies the web developer that *general_-log* for the database is enabled.

Step 2: The web developer exercises the website in a non-malicious manner. For example, if there is a login page the web developer logs in as a registered user. Figure 6.2 illustrates the web developer exercising the login page in a non-malicious manner. Listing 6.1 illustrates the execution of the `SELECT` command that was written to the general log file for the database. Once the web developer has fully exercised every page within the web application, the web developer notifies the DBMS administrator.

Step 3: : The DBMS disables *general_log* and then provides the logs for analysis.

Step 4: Once the logs are provided the script `BuildDBQueries` is executed.
The `BuildDBQueries` script removes duplicate entries from the log file. The script keeps the entries in the log file that start with Query. As stated in Chapter 12 - Section 12.3 the referrer page must also be an entry into the log file. Listing 6.2 illustrates the referrer line that is required for the `BuildDBQueries` to execute correctly.

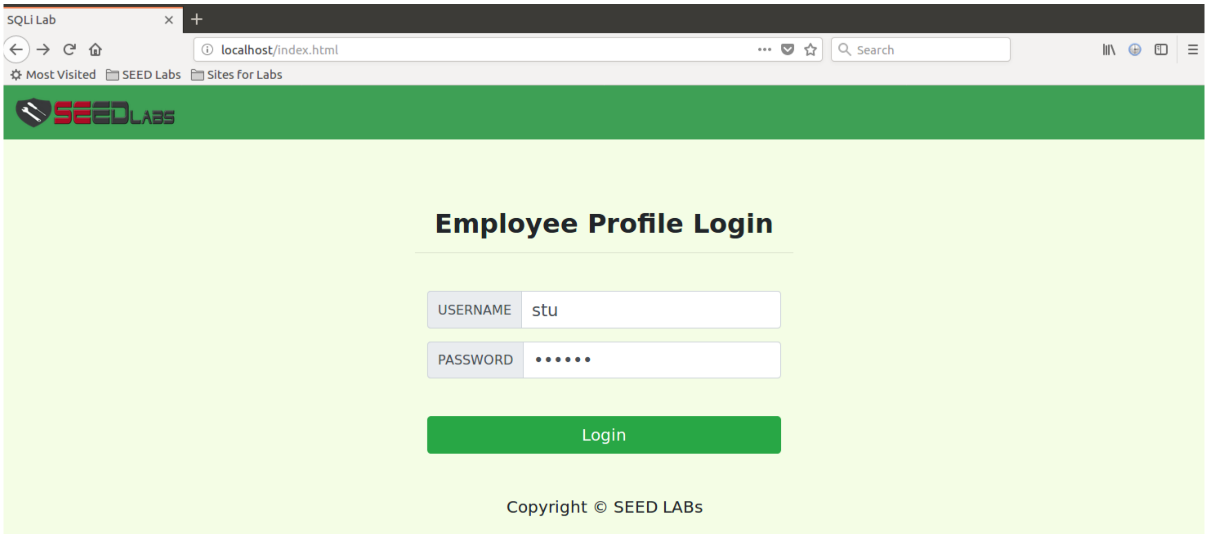


Figure 6.2: The SEED Web Application: This figure is an example of the SEED web application being exercised in a non-malicious manner [1] Version: February 2018

Listing 6.1: Web Developer Exercises Web Application: The resulting SQL query resulting from a non-malicious login in to the web application, exercised by the web developer

```
2018-06-04T23:38:44.913532Z 150 Query SELECT id, name, eid, salary,
    birth, ssn, phoneNumber, address, email,nickname,Password
       FROM credential
       WHERE name= 'stu' and Password='36
    da2c7673be09d05daa028d25741b0d186913d5'
```

Listing 6.2: The referrer query required by the toolset of this dissertation

```
2018-06-04T23:38:44.913752Z 150 Query INSERT INTO track(ref) VALUES
    ('page_name = unsafe_home.php')
```

Step 5: After the database log file has been cleaned, the script `Dynamic2HPol.py` executes, reading the clean database log, and creates a dynamic Python file. The dynamic Python file is executed creating the HPol security model and a HERMES file. The Python script named `example.py` creates the two files `db-example-hpol.pdf`, shown in Figure 6.3, and `db-example-hpol.hermes`. Listing 6.3 illustrates a portion of the `example.py` Python code to create `db-example-hpol.pdf`.

The created HERMES file follows the specification as defined in Section 4.3. Listing 6.4 illustrates an Entity Head and Entity Body from the HERMES grammar. In this example the Entity Head is a first order predicate of Node and Policy.

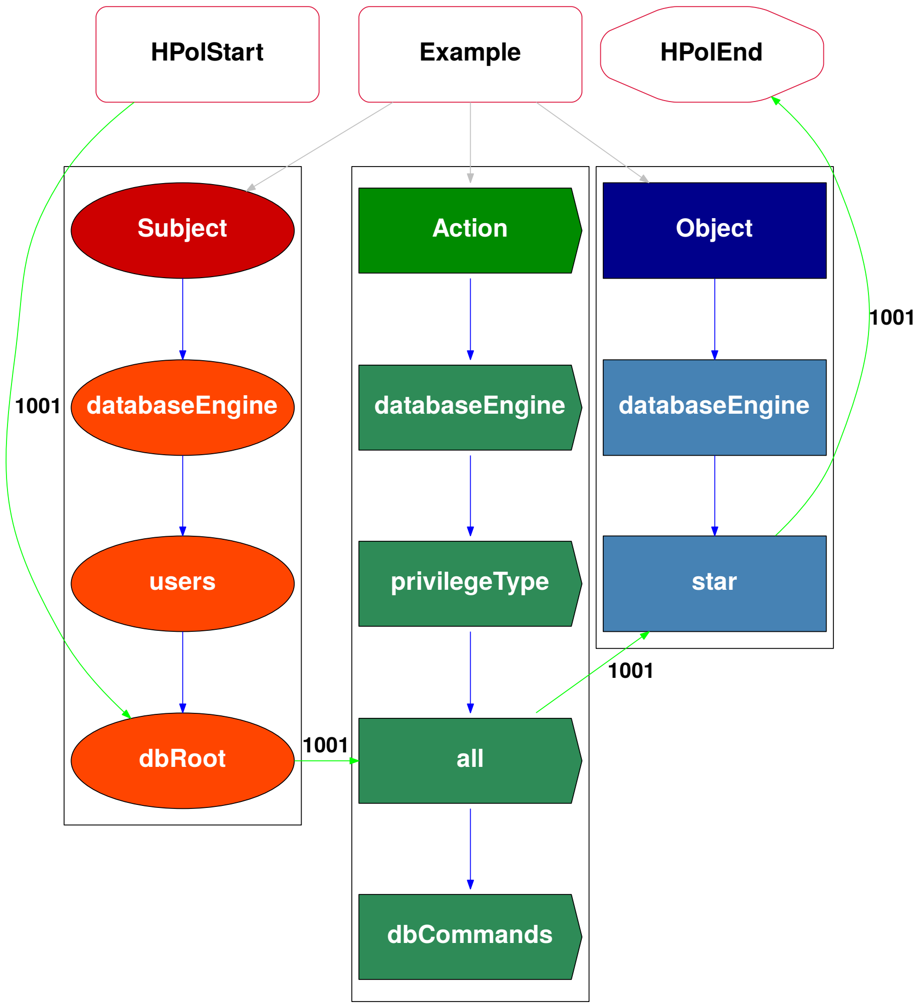


Figure 6.3: Example Dynamic HPol (db-example-hpol.pdf): The HPol security model generated from the clean database logs. This security model represents the web application as a non-least privilege model.

Listing 6.3: Example Python Code: The Python code from `example.py` that creates the `db-example-hpol.hermes` file

```
#-------------------- Create Dot
hpol.createDot('example')

#-------------------- Convert To Hermes
HPol2Hermes.convert2Hermes(hpol)
```

Listing 6.4: Example Dynamic HERMES: A portion of the HERMES generated from the clean database logs. This file represents the Nodes and Policies of the non-least privilege web application.

```
Node: Example
{
    FQN: Namespace.Domain.Example;
    Description: "HPol Root Node";
    Path: "Example";
    Type: "HPolRoot";
}.

Policy: PID_1001
{
    FQN: Namespace.Domain.PID_1001;
    Description: 'Policy';
    Status: ENABLED;
    AbsolutePath: [HPolStart, Example.Subjects.databaseEngine.users.
   dbRoot, Example.Actions.databaseEngine.privilegeType.all,
   Example.Objects.databaseEngine.star, HPolEnd];
    RelativePath: [HPolStart, dbRoot, all, star, HPolEnd];
}.
```

Step 6: Once the non-least privilege HERMES file is created it needs to be converted into a Prolog Knowledge Base (PKB). Recall that HERMES is a specification that contains at least one Entity. Each Entity is comprised of an Entity Head and an Entity Body. Each Entity Head and Entity Body from the HERMES grammar becomes a set of unique Prolog statements. A Prolog fact is a predicate expression that makes a declarative statement about the problem domain [65]. All Prolog sentences must end with a period. An example of a simple prolog fact is “`likes(alice, bob)".` Which is read as “alice likes bob”. In this simple example, alice and bob are not quoted since each is atom. A Prolog data structure can be one of the following types:

- A string atom, for example ,‘`This is a string`’ or `"This is also a string"`.
- A symbol atom, for example, `alice` and `bob` are symbols. Prolog symbols must start with a lower case letter and then a symbol can include digits and the underscore character.
- An empty list atom, for example, `[]`. Lists that contain data are not considered atoms.
- A list, for example, `[1, 2, 3]`. A Prolog list is a comma-separated sequence of items, between square brackets.

In this case, LP HERMES is converted to a LP Prolog Knowledge Base (LP PKB).

Step 7: The automation process for converting from non-least privilege to least privilege is completed via a PKB. After the HERMES conversion to a PKB, the Prolog program `XSB` is executed. The PKB file is loaded via `XSB` and converted to a LP PKB.

An alternative option of converting from HERMES to LP HERMES (not shown in Figure 6.1) is to convert the non-least privilege HERMES output file via a Python script. The Python script `Hermes2LPHermes` performs such a conversion.

Step 8 a): The LP PKB is converted to an LP HERMES file via a Python script `Prolog2Hermes`. The web developer reviews the LP HERMES file to ensure it is complete and correct. If the file is not correct or complete the web developer can edit the HERMES file. Once the HERMES file is edited, it is considered to be a non-least privilege specification. The process of converting the HERMES file to LP HERMES starts again from **Step 6**.

Step 8 b): Once the LP HERMES file has been verified as correct and complete, then the LP PKB is converted to an LP HPol security model via a Python script `Prolog2HPol`. Since the HERMES was verified as correct and complete this is the final version of the HPol security model that the web developer will use to modify the web application code to enforce the Principle of Least Privilege.

Step 9: Once the LP HERMES file has been verified as correct and complete, the LP HERMES file is passed to the Python script `Hermes2SQL`, which constructs a new text file containing the least privilege SQL (LP SQL) commands. These LP SQL commands will create a new user, and grant the appropriate permissions on the appropriate table. This LP SQL file is provided to the DBMS administrator. Note the GRANT SQL commands and the UPDATE commands in Listing 6.5. The GRANT commands limit the permissions of the new user to a certain database and a certain table within that database. Since the new user is created in the global database *mysql* and the global table *user*, the UPDATE commands allow the user to execute commands on the databases specified with the grant commands.

Listing 6.5: Example Dynamic LP SQL: The SQL commands to create a new user and to assign that new user the appropriate permissions for the appropriate tables.

```
CREATE USER IF NOT EXISTS 'newuser'@'localhost' identified by '
    passwd';
GRANT SELECT on database.table to 'newuser'@'localhost';
GRANT UPDATE on database.table2 to 'newuser'@'localhost';
FLUSH PRIVILEGES;
```

Step 10: The DBMS administrator changes the new user's password in the provided LP SQL file, if desired. (The new user's password was set to 'passwd' by default.) The DBMS administrator imports the LP SQL file.

Once LP SQL has been imported by the DBMS administrator, the web developer must refactor the code and then the web application can be tested for non-malicious and malicious operation. The process of creating the LP SQL file and importing it is explained in Chapter 7. The process of systematically refactoring the web application via the LP HPol security model is explained in Chapter 8.

6.2 Applied Case Study: SEED Labs - Inferring Non-Least Privilege

Security Education (SEED) Labs is similar to Mutillidae, in that SEED Labs is a set of hands-on labs for teaching security education. SEED Labs contains a small SQL injection attack lab. The SEED Labs SQL Injection is a deliberately vulnerable Web application. SEED Labs supplies a Virtual Box image [66]. The image contains an Apache Web Server, plus PHP for middleware and a MySQL DBMS back-end. SEED Labs may be installed on Linux, Windows, or MacOS using a LAMP, WAMP, or XAMMP application stack [1].

SEED Labs is a much smaller application than Mutillidae. The SEED Labs SQL injection lab contains the following four web pages.

- `index.html` (index)
- `unsafe_home.php` (home)

- `unsafe_edit_frontend.php` (frontend)
- `unsafe_edit_backend.php` (backend)

The `index` page displays the `home` page. The user attempts to login using the form from the `index` page. Once the user clicks on the `login` button the `home.php` page is executed. This `home` page contains the database code to execute the database query. Appendix B displays the full raw SEED log. Listing 6.6 displays a portion of the database log that is cleaned by `BuildDBQueries`. Figure 6.4 displays the standard SEED Labs login page.

Listing 6.6: SEED Log Header: The header to the seed.log file. These headers are ignored since it does not contain SQL queries.

```
/usr/sbin/mysqld, Version: 5.7.19-0ubuntu0.16.04.1 ((Ubuntu)).
    started with:
Tcp port: 3306  Unix socket: /var/run/mysqld/mysqld.sock
Time                 Id Command    Argument
2018-06-04T23:38:04.148578Z 149 Query set global general_log = 'ON'
```

The SEED Labs PHP code did not contain information on the referrer page. The referrer page was added to the PHP code to allow for proper execution of the Least Privilege toolset. The referrer page was passed as a PHP `$_SESSION` variable. Listing 6.7 illustrates the PHP code added to each PHP page. This code was used to obtain the referrer page.

In the SEED web application [1] version: February 2018, the PHP code for the `home` page connects to the MySQL database `Users`. Once the connection is verified the `SELECT` database query is executed. Since the database user is `root`, then the connection is a non-least privilege connection.

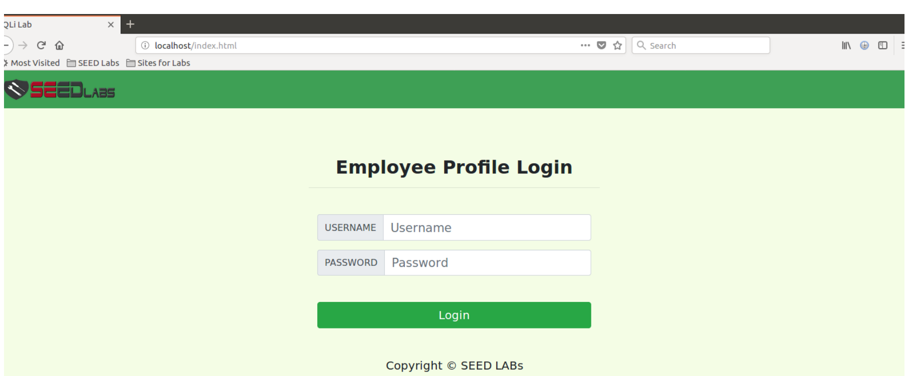


Figure 6.4: SEED Labs Login: The login page as displayed on `index.html` for the SEED Labs SQL Injection lab. [1] Version: February 2018

Listing 6.7: Referrer Page: The PHP code added to the SEED Labs to capture the referrer page.

```
    <?php
    session_start();
    $incoming = $_SESSION['page']; // who called this page?
    $_SESSION['page'] = 'unsafe_home.php'; // setting the name for
  the next page
    $conn = getDB();
    $sqlr = "INSERT INTO track (ID, ref) VALUES (NULL, 'page_name
 = ', $incoming );";
    $conn->query($sqlr)
```

Listing 6.8 displays the MySQL connection and the `SELECT` database query from the `home` page. The successful query returns a JavaScript Object Notation (JSON) object which is parsed to fill in the data on the `frontend` page.

Similar to Mutillidae the general_log in the database was enabled, and the web application was exercised in a non-malicious manner. The log file `seed.log` was produced from the MySQL database. Executing the Python script `./Dynamic2HPol seed.log` performs the following steps:

Step 1: The Python script `BuildDBQueries` cleans `seed.log` and stores the results of the clean log internal as an HPol object. `BuildDBQueries` also stores the results as a text file for verification by the DBMS administrator. Listing 6.9 illustrates the data structures gleaned from the SEED Labs database log.

Listing 6.8: SEED Labs Unsafe DB Query: The SQL query from the `unsafe_home.php`. This query user the `root` user to execute the query.

```
      $dbhost="localhost";
      $dbuser="root";
      $dbpass="seedubuntu";
      $dbname="Users";
      // Create a DB connection
      $conn = new mysqli($dbhost, $dbuser, $dbpass, $dbname);

    $sql = "SELECT id, name, eid, salary, birth, ssn, phoneNumber,
  address, email,nickname,Password
    FROM credential
    WHERE name= '$input_uname' and Password='$hashed_pwd'";
    if (!$result = $conn->query($sql)) {
      echo "</div>";
      echo "</nav>";
      echo "<div class='container text-center'>";
      die('There was an error running the query [' . $conn->error
 . ']\n');
      echo "</div>";
    }
    $return_arr = array();
    while($row = $result->fetch_assoc()){
      array_push($return_arr,$row);
    }
```

Listing 6.9: SEED Labs Data Structure: The users, tables, and databases being utilized by the SEED Labs web application, as determined from the database log.

```
pages
unsafe_home.php
unsafe_edit_frontend.php
unsafe_edit_backend.php

users
root@localhost

dbCommands
SELECT
UPDATE
tables
Users-['credential']
```

Step 2: Once the data structures are determined, the Python script `WriteDynamicHPol` is executed. This script writes the standard HPol header. This header is hardcoded except for the namespace, domain, and web application name. The header is hardcoded because every HPol model must contain Subject, Action, and Object. Listing 6.10 illustrates the HPol Subjects. Listing 6.11 illustrates the HPol Actions, Listing 6.12 illustrates the HPol Objects, and Listing 6.13 illustrates the policies. Furthermore, the filesystem and web server are presumed to be located in the standard locations, so these values are also hardcoded. The execution of `WriteDynamicHPol` creates the Python script `db-seed-hpol`.

Step 3: The file `db-seed-hpol.hermes` is passed as an input to the Python script `Hermes2Prolog` which produces the Prolog Knowledge Base (PKB) named `db-seed-hpol.pro`. Listing 6.14 illustrates the first two nodes from the HERMES file and the first policy in the HERMES file.

Listing 6.10: SEED Labs HPol: Subjects - The dynamic creation of HPol data structure as determined from the SEED Labs database log.

```
hpol.addNode(type='subject', name='databaseEngine', path='/'.join(['
    db', 'Subject']))
hpol.addNode(type='subject', name='users', path='/'.join(['db', '
    Subject','databaseEngine']))
subUser0 = hpol.addNode(type='subject', name='rootATlocalhost', path
    ='/'.join(['db', 'Subject','databaseEngine', 'users']))
hpol.addNode(type='subject', name='filesystem', path='/'.join(['db',
     'Subject']))
hpol.addNode(type='subject', name='var', path='/'.join(['db', '
    Subject', 'filesystem']))
hpol.addNode(type='subject', name='www', path='/'.join(['db', '
    Subject', 'filesystem', 'var']))
```

Listing 6.11: SEED Labs HPol: Actions - The dynamic creation of HPol data structure as determined from the SEED Labs database log.

```
actDBEngine_path = hpol.addNode(type='action', name='databaseEngine', path='/'.join(['db', 'Action']))
actPrivilegeType_path = hpol.addNode(type='action', name='privilegeType', path='/'.join(['db', 'Action','databaseEngine']))
actPrivilegeTypeAll_path = hpol.addNode(type='action', name='all', path='/'.join(['db', 'Action','databaseEngine', 'privilegeType']))
actUser0 = hpol.addNode(type='action', name='select', path='/'.join(['db', 'Action','databaseEngine', 'privilegeType', 'all']))
actUser1 = hpol.addNode(type='action', name='update', path='/'.join(['db', 'Action','databaseEngine', 'privilegeType', 'all']))
```

Listing 6.12: SEED Labs HPol: Objects - The dynamic creation of HPol data structure as determined from the SEED Labs database log.

```
hpol.addNode(type='object', name='databaseEngine', path='/'.join(['db', 'Object']))
hpol.addNode(type='object', name='star', path='/'.join(['db', 'Object', 'databaseEngine']))
hpol.addNode(type='object', name='starDotStar', path='/'.join(['db', 'Object', 'databaseEngine', 'star']))
hpol.addNode(type='object', name='UsersDotStar', path='/'.join(['db', 'Object', 'databaseEngine', 'star', 'starDotStar']))
hpol.addNode(type='object', name='tables', path='/'.join(['db', 'Object', 'databaseEngine', 'star', 'starDotStar', 'UsersDotStar']))
hpol.addNode(type='object', name='credential', path='/'.join(['db', 'Object', 'databaseEngine', 'star', 'starDotStar', 'UsersDotStar','tables']))
```

Listing 6.13: SEED Labs HPol: Policies - The dynamic creation of HPol data structure as determined from the SEED Labs database log.

```
dbSub_path = '/'.join(['db', 'Subject', 'databaseEngine', 'users', 'rootATlocalhost'])
dbAct_path = '/'.join(['db', 'Action', 'databaseEngine', 'privilegeType', 'all', 'select'])
dbObj_path = '/'.join(['db', 'Object', 'databaseEngine', 'star', 'starDotStar', 'UsersDotStar', 'tables', 'credential'])
fsSub_page = '/'.join(['db', 'Subject', 'filesystem', 'var', 'www', 'html', 'seed', 'unsafe_home.php'])

ppid2 = hpol.createEmptyPolicyPath(type='Database Policy')
hpol.addStartLinkToPolicyPath(ppID = ppid2, toNode=fsSub_page)
hpol.addLinkToPolicyPath(ppID = ppid2, fromNode=fsSub_page, toNode=dbSub_path)
hpol.addLinkToPolicyPath(ppID = ppid2, fromNode=dbSub_path, toNode=dbAct_path)
hpol.addLinkToPolicyPath(ppID = ppid2, fromNode=dbAct_path, toNode=dbObj_path)
hpol.addEndLinkToPolicyPath(ppID = ppid2, fromNode=dbObj_path)
```

Listing 6.14: SEED Labs PKB: The dynamic creation of Prolog Knowledge Base as interpreted from the non-least privilege HERMES file.

```
node("db-seed-hpol.hermes", seed_hpol_db, description, "HPol Root
    Node").
node("db-seed-hpol.hermes", seed_hpol_db, path, "db").
node("db-seed-hpol.hermes", seed_hpol_db, type, "HPolRoot").
node("db-seed-hpol.hermes", seed_hpol_db_Subject, description, "
    Subject").
node("db-seed-hpol.hermes", seed_hpol_db_Subject, path, "db/Subject
    ").
node("db-seed-hpol.hermes", seed_hpol_db_Subject, type, "Subject").
policy("db-seed-hpol.hermes", seed_hpol_PIDUNSC1001, description, '
    Database Policy').
policy("db-seed-hpol.hermes", seed_hpol_PIDUNSC1001, status, eNABLED
    ).
policy("db-seed-hpol.hermes", seed_hpol_PIDUNSC1001, relativePath, [
    hPolStart, unsafeUNSChomeDOTphp, rootATlocalhost, select,
    credential, hPolEnd]).
policy("db-seed-hpol.hermes", seed_hpol_PIDUNSC1003, description, '
    Database Policy').
policy("db-seed-hpol.hermes", seed_hpol_PIDUNSC1003, status, eNABLED
    ).
policy("db-seed-hpol.hermes", seed_hpol_PIDUNSC1003, relativePath, [
    hPolStart, unsafeUNSCeditUNSCbackendDOTphp, rootATlocalhost,
    update, credential, hPolEnd]).
```

Step 4: The file `db-seed-hpol.pro` is loaded into XSB Prolog. The least privilege algorithm converts the non-least privilege PKB to a least privilege PKB (LP PKB). The LP PKB file in this case study is named `db_seed_hpol.pro`.

Step 5: The Python script `db-seed-hpol` executes and produces two files as output. The first file `db-seed-hpol.pdf` is the graphical representation of the non-least privilege SEED Labs web application. This graphical representation is created in the form of a directed acyclic graph (DAG). The second file `db-seed-hpol.hermes` is the HERMES grammar representation of the non-least privilege SEED Labs web application.

Step 6: The Python script `Prolog2Hermes` converts the LP PKB file `db_seed_hpol.pro` into a HERMES file named `db_seed_hpol.hermes`. This file is examined by the web developer to determine if the HERMES file is correct and complete. The web developer can make modifications to the HERMES file. If modifications are made, then the HERMES file must be rerun through the process by converting it to a non-least privilege Prolog file.

Step 7: Once the `db_seed_hpol.hermes` file is determined to be correct and complete, the Python script `Prolog2HPol` is executed. This execution creates two Least Privilege HPol security models (LP HPOL) the first file is named `db_seed_hpol.py` and the second file is named `db_seed_hpol.pdf`. The Python file `db_seed_hpol.py` creates the HPol security model. The file `db_seed_hpol.pdf` is the DAG representation of LP HPol security model. Figure 6.5 illustrates the full LP DAG for the SEED Labs web application.

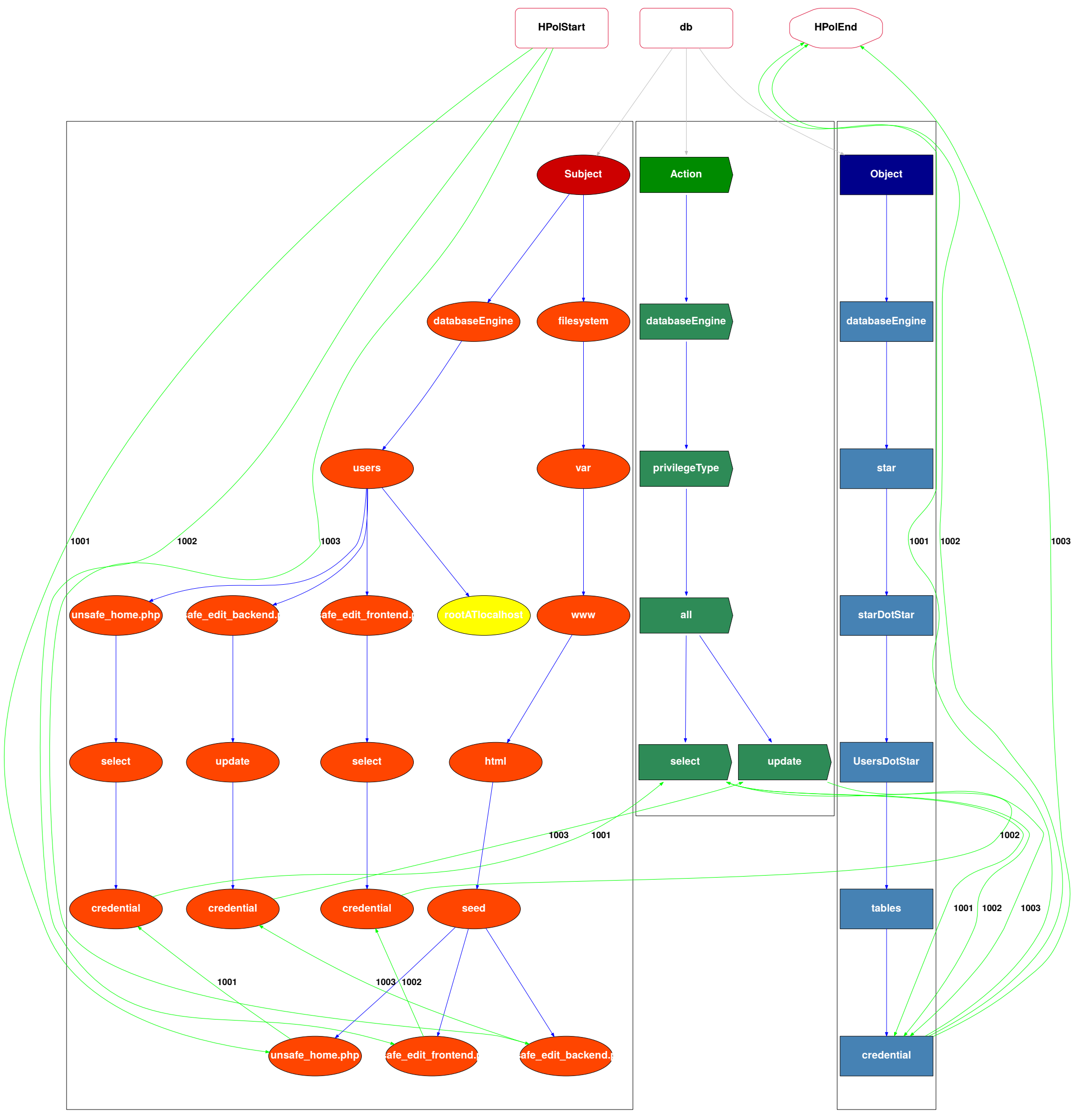


Figure 6.5: Example LP HPol: The LP HPol security model generated from the LP PKB. This security model represents the web application as a least privilege model.

The root database user remains in the DAG; however, it is not referenced by any policy. The Subject `rootATlocalhost` is represented as the clear node in the Subject HPol security model.

6.3 Applied Case Study: Mutillidae - Inferring Non-Least Privilege

Recall from Section 4.7 that OWASP Mutillidae II (version: 2.6.42) is a deliberately vulnerable Web application. Mutillidae II may be used by developers to learn secure Web coding practices. It uses a Web Server, such as Apache, plus PHP for middleware and a DBMS back-end, such as MySQL or MariaDB. Mutillidae II may be installed on Linux, Windows, or MacOS using a LAMP, WAMP, or XAMMP application stack [14].

To initiate the process of converting Mutillidae from non-least privilege to least privilege, the general_log of the MySQL Mutillidae database was enabled. Once the logs were enabled, the Mutillidae web application was systematically exercised in a non-malicious manner.

The full raw Mutillidae log is displayed in Appendix A. Listing 6.15 illustrates the header of the Mutillidae log file. Since the header does not contain SQL commands it is completely ignored by the `BuildDBQueries`. Listing 6.16 illustrates repeated log entries in the file `mutillidae.log`. These repeated entries are reduced to a single entry.

Listing 6.15: Mutillidae Log Header: The header to the `mutillidae.log` file. These headers are ignored since it does not contain SQL queries.

```
/usr/sbin/mysqld, Version: 5.7.20-0ubuntu0.16.04.1 ((Ubuntu)).
   started with:
Tcp port: 3306  Unix socket: /var/run/mysqld/mysqld.sock
Time                 Id Command    Argument
2017-12-26T18:59:18.029648Z     7 Quit
```

Listing 6.16: Mutillidae Log: The SQL queries from the Mutillidae web application database log. These queries were created when the Mutillidae web application was exercised in a non-malicious manner.

```
1  2017-12-26T18:59:28.522673Z     8 Connect   root@localhost on  using
       Socket
2  2017-12-26T18:59:28.523446Z     8 Init DB   nowasp
3  2017-12-26T18:59:28.523505Z     8 Query SELECT 'test connection'
4  2017-12-26T18:59:28.523577Z     8 Query SELECT cid FROM blogs_table
5  2017-12-26T18:59:28.528773Z     8 Quit
6  2017-12-26T18:59:28.529031Z     9 Connect   root@localhost on  using
       Socket
7  2017-12-26T18:59:28.529095Z     9 Init DB   nowasp
8  2017-12-26T18:59:28.531037Z    10 Connect   root@localhost on  using
       Socket
9  2017-12-26T18:59:28.531098Z    10 Init DB   nowasp
10 2017-12-26T18:59:28.531412Z    11 Connect   root@localhost on  using
       Socket
11 2017-12-26T18:59:28.531465Z    11 Init DB   nowasp
12 2017-12-26T18:59:28.531780Z    12 Connect   root@localhost on  using
       Socket
```

In this example the duplicated "`Connect root@localhost on using Socket`" on lines 1, 6, 8, 10, 12, becomes a single entry in the cleaned log file that is used by `BuildDBQueries`. Executing the Python script `./Dynamic2HPol mutillidae.log` performs the following actions:

Step 1: The Python script `BuildDBQueries` cleans `mutillidae.log` and stores the results of the clean log internally as an HPol object. `BuildDBQueries` also stores the results as a text file for verification by the DBMS administrator. Listing 6.17 illustrates the data structures gleaned from the Mutillidae database log.

Listing 6.17: Mutillidae Data Structure: The users, tables, and databases being utilized by the Mutillidae web application, as determined from the database log.

```
show-log.php
add-to-your-blog.php
view-someones-blog.php
test.php
apage.php

users
root@localhost

databases
nowasp

dbCommands
SELECT
INSERT
```

Step 2: Once the data structures are determined, the Python script `WriteDynamicHPol` is executed. This script writes the standard HPol header. This header is hardcoded except for the namespace, domain, and web application name. The header is hardcoded for filesystem and web server locations, and because every HPol model contains a Subject, Action, and Object. The execution of `WriteDynamicHPol` creates the Python script `db-mutillidae-hpol.py`. Listing 6.18 illustrates the HPol Subjects, Listing 6.19 the HPol Actions, Listing 6.20 the HPol Objects and Figure 6.21 illustrates the HPol Policies.

Listing 6.18: Mutillidae HPol: Subjects - From the file: db_mutillidae_hpol.py - The dynamic creation of HPol data structure as determined from the Mutillidae database log.

```
hpol.addNode(type='subject', name='databaseEngine', path='/'.join(['
    db', 'Subject']))
hpol.addNode(type='subject', name='users', path='/'.join(['db', '
    Subject','databaseEngine']))
subUser0 = hpol.addNode(type='subject', name='rootATlocalhost', path
    ='/'.join(['db', 'Subject','databaseEngine', 'users']))
hpol.addNode(type='subject', name='filesystem', path='/'.join(['db',
     'Subject']))
hpol.addNode(type='subject', name='var', path='/'.join(['db', '
    Subject', 'filesystem']))
hpol.addNode(type='subject', name='www', path='/'.join(['db', '
    Subject', 'filesystem', 'var']))
hpol.addNode(type='subject', name='html', path='/'.join(['db', '
    Subject', 'filesystem', 'var', 'www']))
hpol.addNode(type='subject', name='mutillidae', path='/'.join(['db',
     'Subject', 'filesystem', 'var', 'www', 'html']))
```

Listing 6.19: Mutillidae HPol: Actions - From the file: db_mutillidae_hpol.py - The dynamic creation of HPol data structure as determined from the Mutillidae database log.

```
actDBEngine_path = hpol.addNode(type='action', name='databaseEngine
    ', path='/'.join(['db', 'Action']))
actPrivilegeType_path = hpol.addNode(type='action', name='
    privilegeType', path='/'.join(['db', 'Action','databaseEngine'])
    )
actPrivilegeTypeAll_path = hpol.addNode(type='action', name='all',
    path='/'.join(['db', 'Action','databaseEngine', 'privilegeType
    ']))
actUser0 = hpol.addNode(type='action', name='select', path='/'.join
    (['db', 'Action','databaseEngine', 'privilegeType', 'all']))
actUser1 = hpol.addNode(type='action', name='insert', path='/'.join
    (['db', 'Action','databaseEngine', 'privilegeType', 'all']))
```

Listing 6.20: Mutillidae HPol: Objects - From the file: db_mutillidae_hpol.py - The dynamic creation of HPol data structure as determined from the Mutillidae database log.

```
hpol.addNode(type='object', name='databaseEngine', path='/'.join(['
    db', 'Object']))
hpol.addNode(type='object', name='star', path='/'.join(['db', '
    Object', 'databaseEngine']))
hpol.addNode(type='object', name='starDotStar', path='/'.join(['db',
     'Object', 'databaseEngine', 'star']))
hpol.addNode(type='object', name='nowaspDotStar', path='/'.join(['db
    ', 'Object', 'databaseEngine', 'star', 'starDotStar']))
hpol.addNode(type='object', name='tables', path='/'.join(['db', '
    Object', 'databaseEngine', 'star', 'starDotStar', 'nowaspDotStar
    ']))
hpol.addNode(type='object', name='blogs_table', path='/'.join(['db',
     'Object', 'databaseEngine', 'star', 'starDotStar', '
    nowaspDotStar','tables']))
hpol.addNode(type='object', name='accounts', path='/'.join(['db', '
    Object', 'databaseEngine', 'star', 'starDotStar', 'nowaspDotStar
    ','tables']))
```

Listing 6.21: Mutillidae HPol: Policies - From the file: db-mutillidae-hpol.py - The dynamic creation of HPol data structure as determined from the Mutillidae database log.

```
dbSub_path = '/'.join(['db', 'Subject', 'databaseEngine', 'users', '
    rootATlocalhost'])
dbAct_path = '/'.join(['db', 'Action', 'databaseEngine', '
    privilegeType', 'all', 'select'])
dbObj_path = '/'.join(['db', 'Object', 'databaseEngine', 'star', '
    starDotStar', 'nowaspDotStar', 'tables', 'blogs_table'])
fsSub_page = '/'.join(['db', 'Subject', 'filesystem', 'var', 'www',
    'html', 'mutillidae', 'home.php'])
ppid2 = hpol.createEmptyPolicyPath(type='Database Policy')
hpol.addStartLinkToPolicyPath(ppID = ppid2, toNode=fsSub_page)
hpol.addLinkToPolicyPath(ppID = ppid2, fromNode=fsSub_page, toNode=
    dbSub_path)
hpol.addLinkToPolicyPath(ppID = ppid2, fromNode=dbSub_path, toNode=
    dbAct_path)
hpol.addLinkToPolicyPath(ppID = ppid2, fromNode=dbAct_path, toNode=
    dbObj_path)
hpol.addEndLinkToPolicyPath(ppID = ppid2, fromNode=dbObj_path)
```

Step 3: The Python script `db-mutillidae-hpol.py` executes and produces two output files. The first file `db-mutillidae-hpol.pdf`, shown in Figure 6.6 is the graphical representation of the non-least privilege Mutillidae web application. The second file `db-mutillidae-hpol.hermes` is the HERMES grammar representation of the non-least privilege Mutillidae web application. Listing 6.22 illustrates a node and a policy from the non-least privilege HERMES.

Step 4: The file `db-mutillidae-hpol.hermes` is passed as an input to the Python script `Hermes2Prolog` which produces `db-mutillidae-hpol.pro`. Listing 6.23 illustrates a non-least privilege Prolog fact for a node and a policy.

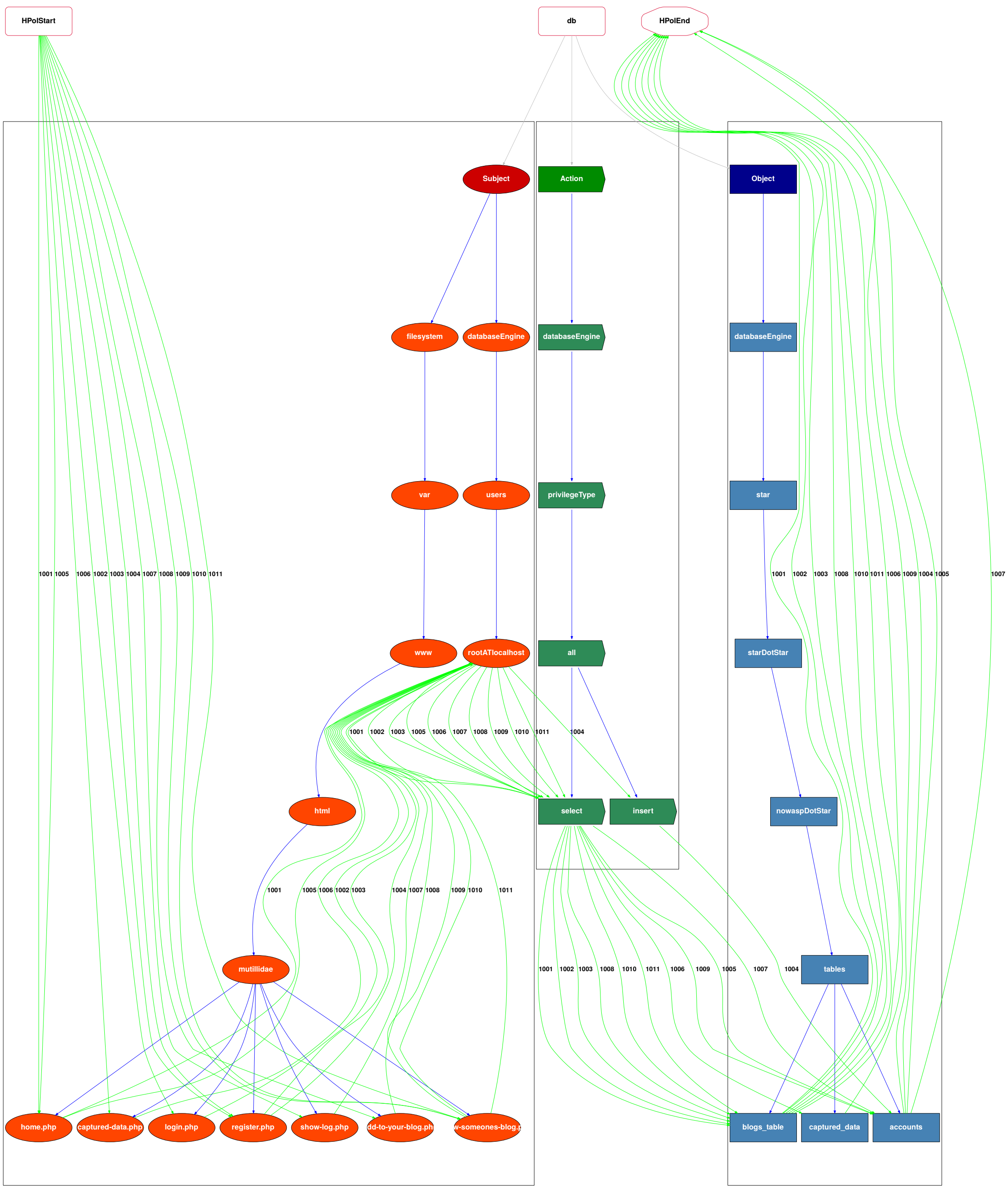


Figure 6.6: Example NLP HPol: The generated non-LP HPol security model.

Listing 6.22: Mutillidae HERMES: From the file: db-mutillidae-hpol.hermes - The dynamic creation of HERMES as interpreted from the non-least privilege HPol file.

```
Node: db
{
    FQN: mutillidae.hpol.db;
    Description: "HPol Root Node";
    Path: "db";
    Type: "HPolRoot";
}.
Policy: PID_1001
{
    FQN: mutillidae.hpol.PID_1001;
    Description: 'Database Policy';
    Status: ENABLED;
    AbsolutePath: [HPolStart, db.Subject.filesystem.var.www.html.
   mutillidae.homeDOTphp, db.Subject.databaseEngine.users.
   rootATlocalhost, db.Action.databaseEngine.privilegeType.all.
   select, db.Object.databaseEngine.star.starDotStar.nowaspDotStar.
   tables.blogs_table, HPolEnd];
}.
```

Listing 6.23: Mutillidae PKB: From the file: db-mutillidae-hpol.pro - The dynamic creation of Prolog Knowledge Base as interpreted from the non-least privilege HERMES file.

```
node("db-mutillidae-hpol.hermes", mutillidae_hpol_db_Subject, path,
   "db/Subject").
node("db-mutillidae-hpol.hermes", mutillidae_hpol_db_Subject, type,
   "Subject").
policy("db-mutillidae-hpol.hermes", mutillidae_hpol_PIDUNSC1010,
   status, eNABLED).
policy("db-mutillidae-hpol.hermes", mutillidae_hpol_PIDUNSC1011,
   relativePath, [hPolStart, viewDASHsomeonesDASHblogDOTphp,
   rootATlocalhost, select, blogsUNSCtable, hPolEnd]).
policy("db-mutillidae-hpol.hermes", mutillidae_hpol_PIDUNSC1007,
   description, 'Database Policy').
policy("db-mutillidae-hpol.hermes", mutillidae_hpol_PIDUNSC1007,
   status, eNABLED).
```

Step 5: The file `db-mutillidae-hpol.pro` is loaded into XSB Prolog. XSB Prolog converts the non-least privilege PKB to a least privilege PKB (LP PKB). The LP PKB file in this case study is named `db_mutillidae_hpol.pro`. Listing 6.24 illustrates the least privilege PKB file while

Step 6: The Python script `Prolog2Hermes` converts `db_mutillidae_hpol.pro` into a HERMES file named `db_mutillidae_hpol.hermes`. This file is examined by the web developer to determine if the HERMES file is correct and complete. The web developer can make modifications to the HERMES file. Modifications require the HERMES file be rerun through the process as a non-least privilege file. Listing 6.25 illustrates the LP HERMES file constructed from the LP PKB file.

Listing 6.24: Mutillidae LP PKB: From the file: db_mutillidae_hpol.pro - The dynamic creation of least privilege PKB as interpreted from the non-least privilege PKB file.

```
node("db_mutillidae_hpol.hermes", mutillidae_hpol_db, description, "
    HPol Root Node").
node("db_mutillidae_hpol.hermes", mutillidae_hpol_db, path, "db").
node("db_mutillidae_hpol.hermes", mutillidae_hpol_db, type, "
    HPolRoot").
node("db_mutillidae_hpol.hermes", mutillidae_hpol_db_Subject,
    description, "Subject").
policy("db_mutillidae_hpol.hermes", mutillidae_hpol_PIDUNSC1001,
    description, 'Database Policy').
policy("db_mutillidae_hpol.hermes", mutillidae_hpol_PIDUNSC1001,
    status, eNABLED).
policy("db_mutillidae_hpol.hermes", mutillidae_hpol_PIDUNSC1001,
    relativePath, [hPolStart, homeDOTphp,
    homeDOTphp_select_blogsUNSCtable, select, blogsUNSCtable,
    hPolEnd]).
```

Listing 6.25: Mutillidae LP HERMES: From the file: db_mutillidae_hpol.hermes - The dynamic creation of LP HERMES as interpreted from the LP PKB file.

```
Node: db
{
     FQN: mutillidae.hpol.db;
     Description: "HPol Root Node";
     Path: "db";
     Type: "HPolRoot";
}.
Policy: PID_1001
{
     FQN: mutillidae.hpol.PID_1001;
     Description: 'Database Policy';
     Status: ENABLED;
     RelativePath: [HPolStart, homeDOTphp, homeDOTphp.select.
    blogs_table, select, blogs_table, HPolEnd];
}.
```

Step 7: Once the `db_mutillidae_hpol.hermes` file is determined to be correct and complete, the Python script `Prolog2HPol` is executed. This execution creates two Least Privilege HPol security models (LP HPOL) the first file is named `db_mutillidae_hpol.py`. This is the Python code to create the HPol security model. The second file is named `db_mutillidae_hpol.pdf` which is the DAG representation of LP HPol security model. Figure 6.7 `db_mutillidae_hpol.pdf` illustrates the full LP DAG for the Mutillidae web application. Although the root database user remains in the DAG, the root user is not referenced by any policy. The Subject `rootATlocalhost` is the empty node in the Subject HPol security model.

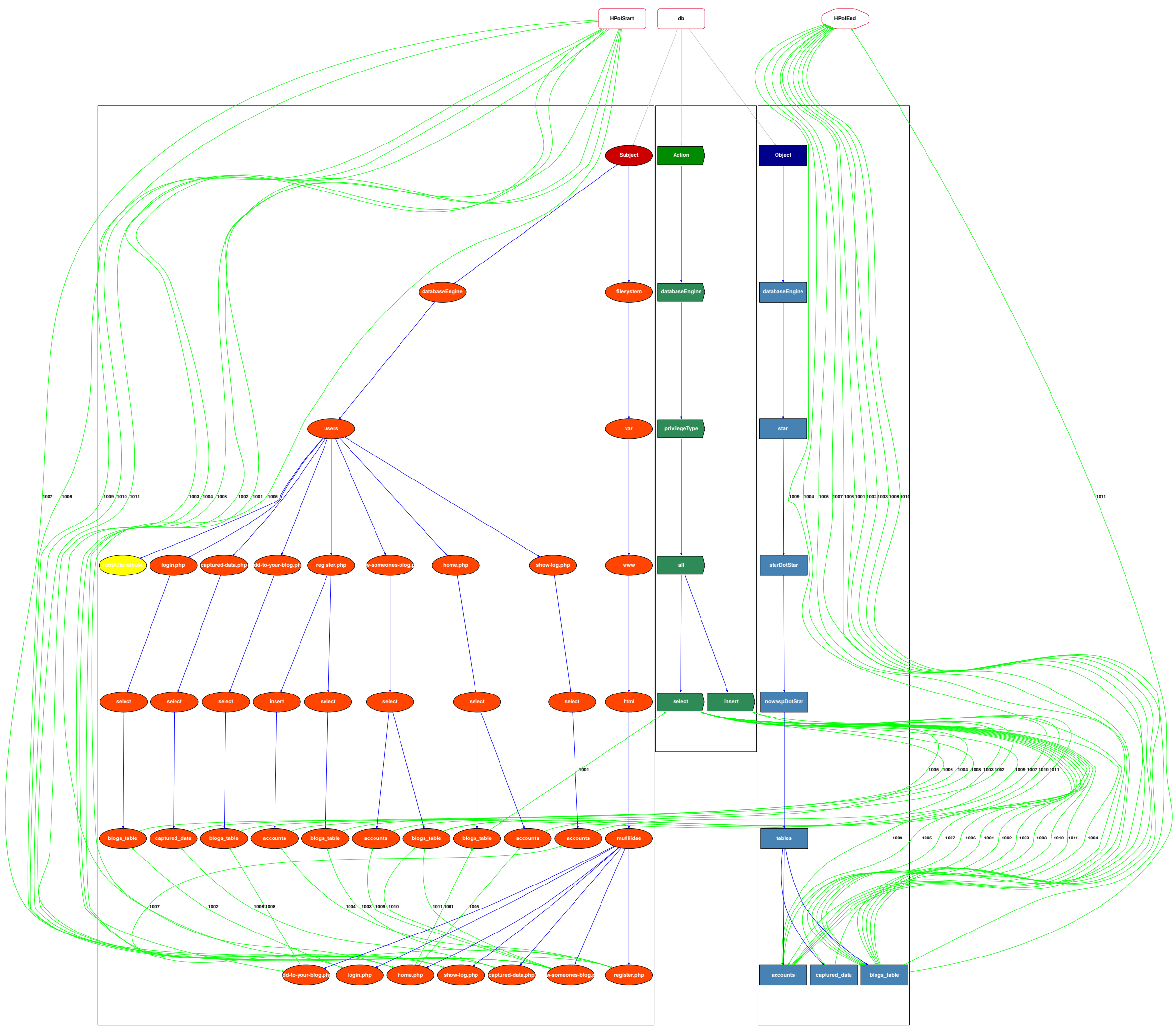


Figure 6.7: Example LP HPol: The LP HPol security model generated from the LP PKB. This security model represents the web application as a least privilege model.

This case study illustrates the repeatable, systematic, semi-automated toolset of scripts that moved the Mutillidae web application from a non-least privilege application to a least privilege application.

Chapter 7: Automating the Transformation of a Web Application to a Least Privilege Implementation

With a robust toolset, the next step in the process is the partial transformation to a least privilege web application via Least Privilege SQL (LP SQL). Section 7.1 explains the process of automating the conversion from LP HERMES to LP SQL. The practical application of LP SQL is applied in two case studies. Section 7.2 explains the application of LP SQL to the Security Education (SEED) Labs. Section 7.3 explains the application of LP SQL to the Mutillidae web application. Figure 7.1 illustrates **Contribution 5: A developed formal, repeatable, and automated approach and associated toolset for determining and applying least privilege permissions at the database level for securing web applications**.

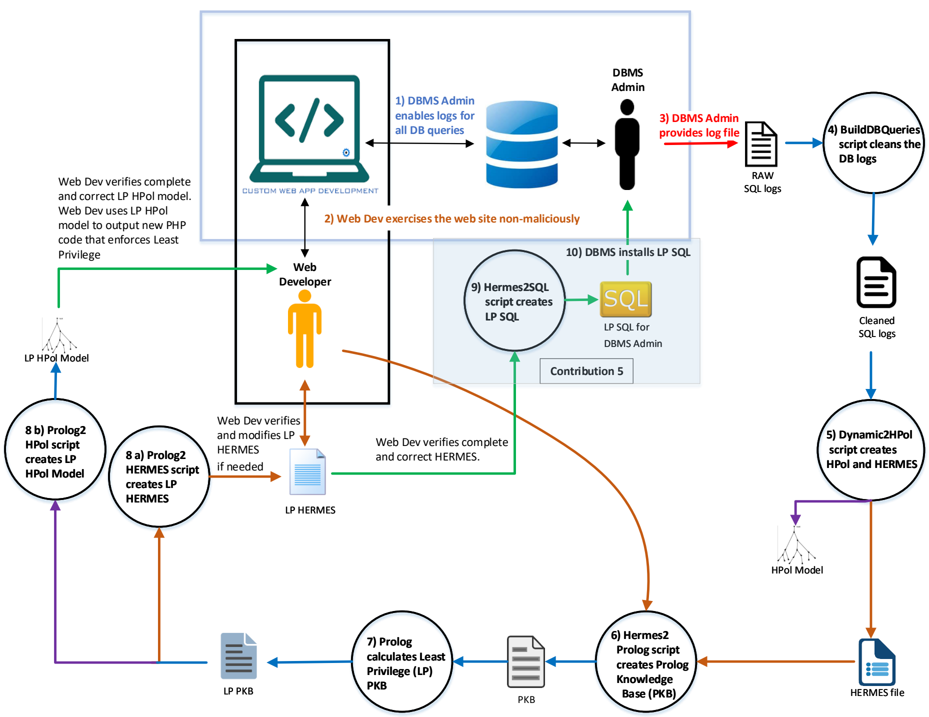


Figure 7.1: Contribution 5: A developed formal, repeatable, and automated approach and associated toolset for determining and applying least privilege permissions at the database level for securing web applications.

7.1 Automated Process for Creating Least Privilege SQL Database Commands

A typical web application contains a single privileged user `root` that has full privileges to the filesystem and the database. Allowing full privileges to the database from a login page, or a similar page, could allow for a malicious user to completely compromise the web application. In order to enforce the POLP the `root` user should be deactivated and less privileged users should be created. Creating such least privileged users is derived via the following steps.

Step 1: Determine the web page that passes the query to the database.

Step 2: Determine the query to the database, such as `SELECT`, `INSERT`, or any of the other SQL commands.

Step 3: Determine the database table that is queried.

Step 4: Concatenate the name of the PHP page, the name of the database query, and the name of the database table to create the new user.

Step 5: Create the new user in the database.

Step 6: Grant the appropriate privileges to the new user.

Based on the HPOL security model, the policy represented in Figure 7.2 illustrates the non-least privilege access to the database. In this example, the page *login.php* issues a database query as the user *rootATlocalhost* using the *SELECT* command.

The HPol security model is also represented as a specification in a HERMES file. In this instance, Figure 7.2 illustrates Policy 1001 as defined by Listing 7.1. The one line in Policy 1001, in Figure 7.2, of: `[HPolStart, loginDOTphp, rootATlocalhost, select, accounts, HPolEnd]` indicates that the policy starts with a link from the start node *HPolStart* to the Subject *login.php*. The policy continues from the Subject *login.php* to the Subject *rootATlocalhost*, to the Action *SELECT*, to the Object *accounts* which is a table in the database, and terminates at the end node *HPolEnd*. Stated another way, the *login.php* page creates a *SELECT* query that is executed by *rootATlocalhost* on the table *accounts*.

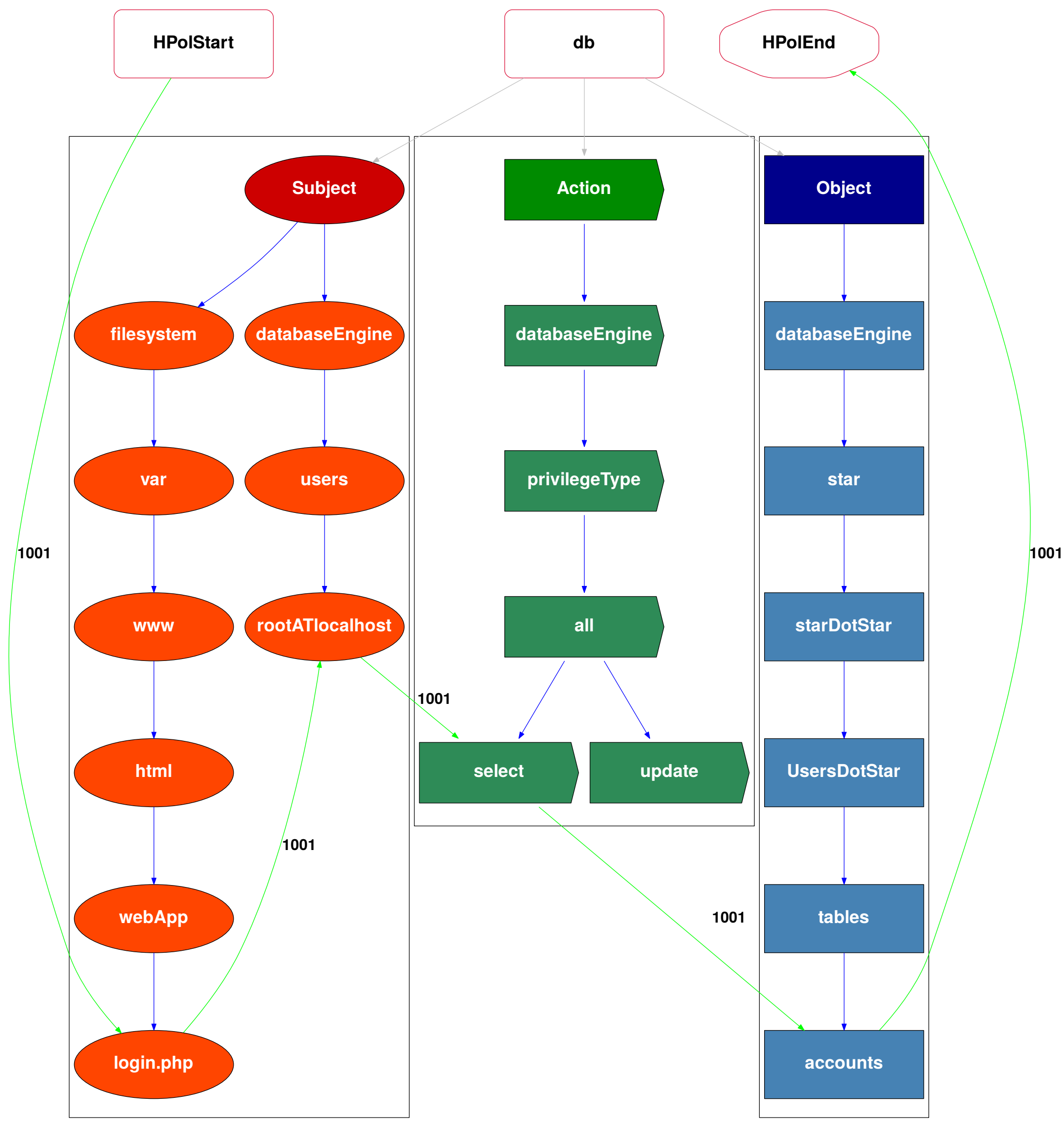


Figure 7.2: Unsecured HPol DB Example: This figure illustrates a non-least privilege database interaction. The HPol model representing the *login.php* queries the database via *rootATlocalhost* which issues the `SELECT` command on the table *accounts*.

Listing 7.1: Non-Least Privilege Policy 1001: A portion of the HERMES that illustrates Policy 1001 indicating the policy starts with a link from the start node to login.php, login.php is linked to *rootATlocalhost*, *rootATlocalhost* is linked to `SELECT`, `SELECT` is linked to the table *accounts*, and the table *accounts* is linked to the end node of the policy.

```
Policy: PID_1001
{
    FQN: example.hpol.PID_1001;
    Description: 'Database Policy';
    Status: ENABLED;
    AbsolutePath: [HPolStart, db.Subject.filesystem.var.www.html.
   webApp.loginDOTphp, db.Subject.databaseEngine.users.
   rootATlocalhost, db.Action.databaseEngine.privilegeType.all.
   select, db.Object.databaseEngine.star.starDotStar.UsersDotStar.
   tables.accounts, HPolEnd];
}.
```

In order to move from non-least privilege to least privilege, a non-privileged set of database users needs to be created. For this dissertation, the specification creates a new database user with the following property. The user will be created based on (1) the name of the web page initiating the query, (2) the name of the SQL command being issued in the database query, and (3) the name of the table being accessed by the query. The creation of the new users occurs after the conversion of NLP HERMES to a non-Least Privilege Prolog Knowledge Base (NLP PKB) and after the execution of Prolog queries on the NLP PKB, yielding LP PKB. Next the LP PKB is converted to LP HERMES and LP HPol.

In this instance Figure 7.3 illustrates the new Least Privilege Policy 1001 as defined by Listing 7.2. The one line in Policy 1001, in Figure 7.3, of: `[HPolStart, loginDOTphp, loginDOTphp.select.accounts, select, accounts, HPolEnd]` indicates the policy starts with a link from the start node *HPolStart* to the Subject *login.php*. The policy continues from the Subject *login.php* to the Subject *loginDOTphp.select.accounts*. The Subject *loginDOTphp.select.accounts* is a non-privileged user that can only execute the the Action `SELECT` on the table *accounts*. The policy continues from the Action `SELECT` to the Object *accounts* which is a table in the database, and terminates at the end node *HPolEnd*. Stated another way, the *login.php* page creates a `SELECT` query executed by the new least privilege user *loginDOTphp.select.accounts*, which has only `SELECT` permissions on the *accounts* table.

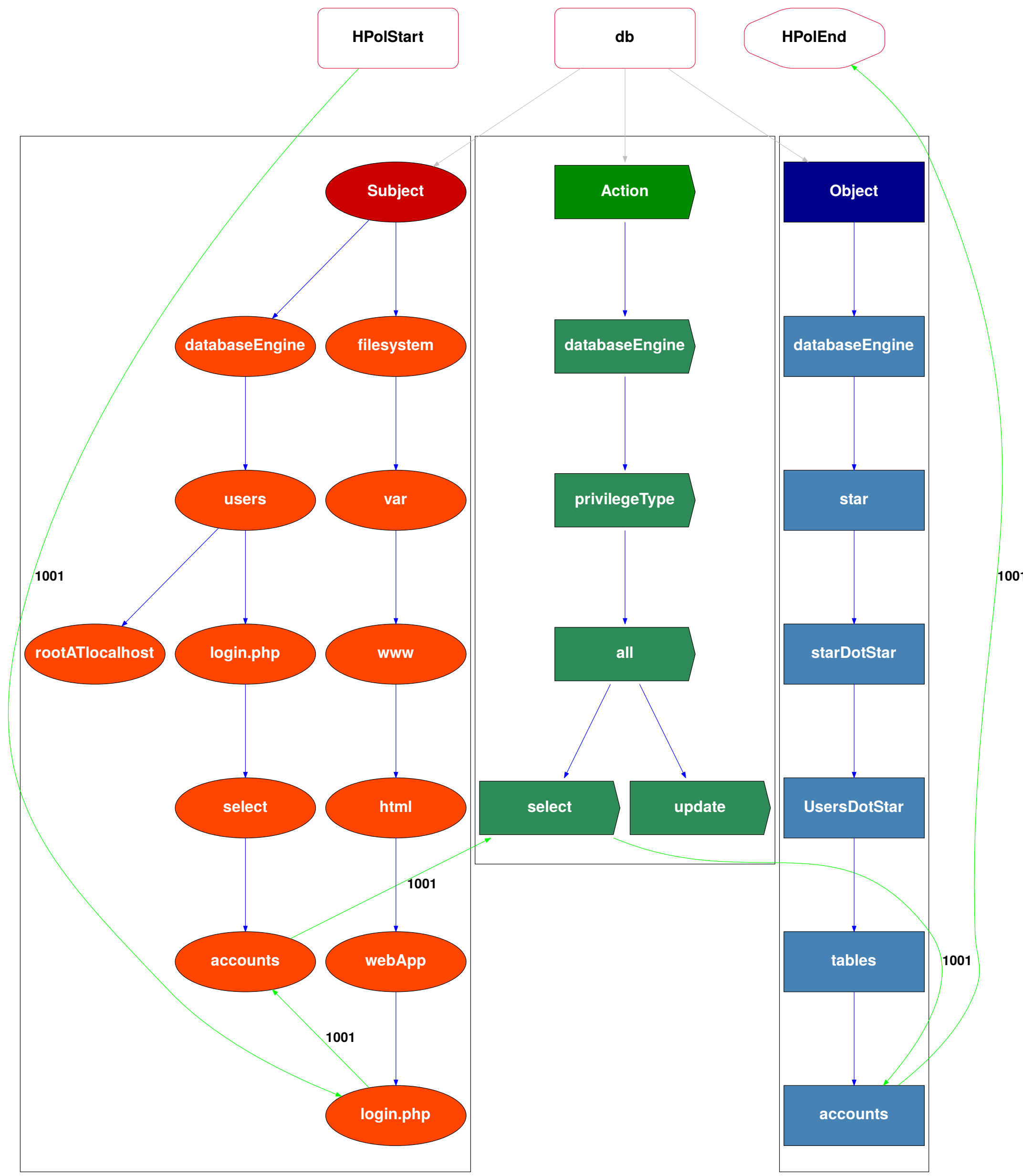


Figure 7.3: Secured HPol DB Example: This figure illustrates a least privilege database interaction. The HPol model representing the *login.php* queries the database via *login.select.accountsATlocalhost* which issues the *SELECT* command on the table *accounts*.

Listing 7.2: Least Privilege Policy 1001: A portion of the HERMES that illustrates Policy 1001 indicating the policy starts with a link from the start node to login.php, login.php is now linked to the non privileged user loginDOTphp.select.accounts, loginDOTphp.select.accounts is linked to SELECT, SELECT is linked to the table accounts, and the table accounts is linked to the end node of the policy.

```
Policy: PID_1001
{
    FQN: example.hpol.PID_1001;
    Description: 'Database Policy';
    Status: ENABLED;
    RelativePath: [HPolStart, loginDOTphp, loginDOTphp.select.
   accounts, select, accounts, HPolEnd];
}.
```

Once the web developer approves the LP HERMES file, the Python script `Hermes2SQL` is executed. This Python script reads the HERMES file and completes the following:

Step 1: Creates a file named the <root node>-<namespace>-<domain>.sql. For this example the file is named *db-example-hpol.sql*.

Step 2: In the *db-example-hpol.sql*, shown in Listing 7.3, file a new database user is created for each new Subject graph from Subject - databaseEngine - users. In this example the new user resides in the HPol security model under Subject - databaseEngine - users - login.php - select - accounts. In the sql file the command issued is `CREATE USER IF NOT EXISTS 'login.select.accounts'@'localhost' IDENTIFIED BY 'passwd';`. Note the *.php* is removed from the username.

Step 3: in the *db-example-hpol.sql* the new user is granted permission for a certain database and a certain table with that database. In the sql file the command issued is `GRANT SELECT on Users.accounts to 'login.select.accounts'@'localhost';`. In this example the database is *Users* and the table is *accounts*.

Step 4: In the *db-example-hpol.sql* file the database is changed to the global database of mysql via the command `USE mysql;`

Listing 7.3: LP SQL (db-example-hpol.sql): The SQL commands that are automatically generated from the correct and complete LP HERMES.

```
CREATE USER IF NOT EXISTS 'login.select.accounts'@'localhost'
    IDENTIFIED BY 'passwd';
GRANT SELECT on Users.accounts to 'login.select.accounts'@'localhost
    ';
FLUSH PRIVILEGES;
```

Step 5: After the new user is created then the database *mysql* and the table *user* need to be updated for the appropriate permissions. In the sql file the command issued is `UPDATE ‘user‘ SET ‘Select_priv‘ = ’Y’ WHERE ‘user‘.‘Host‘ = ’localhost’ AND ‘user‘.‘User‘ = ’login.select.accounts’;`.

With LP SQL, the DBMS administrator can install the SQL statements to move the database from non-least privilege to least privilege.

7.2 Applied Case Study: SEED Labs - Automating a Least Privilege Implementation

This section describes the process of creating LP SQL from LP HERMES for the case study of SEED. Recall from Figure 6.5 a new set of least privilege users were created for the SEED web application. Table 7.1 illustrates the created new least privilege user, for this case study, and Listing 7.4 displays the appropriate SQL commands to create and assign the new users the appropriate permissions.

Table 7.1: Applied SEED LP SQL Users: A manually extracted list, from the HERMES file, of the policy numbers and new users that were added to LP SQL. The LP SQL file is imported by the DBMS administrator as a partial solution to move the web application from non-least privilege to least privilege.

Policy Number	New LP User
Policy 1001	unsafe_home.select.credential
Policy 1002	unsafe_edit_frontend.select.credential
Policy 1003	unsafe_edit_backend.update.credential

Listing 7.4: SEED LP SQL: The SQL commands that are automatically written from the correct and complete LP HERMES for the SEED web application. The SQL file is named db-seed-hpol.sql.

```sql
CREATE USER IF NOT EXISTS 'unsafe_home.select.credential'@'localhost' IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'unsafe_edit_frontend.select.credential'@'localhost' IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'unsafe_edit_backend.update.credential'@'localhost' IDENTIFIED BY 'passwd';
GRANT SELECT on Users.credential to 'unsafe_home.select.credential'@'localhost';
GRANT SELECT on Users.credential to 'unsafe_edit_frontend.select.credential'@'localhost';
GRANT UPDATE on Users.credential to 'unsafe_edit_backend.update.credential'@'localhost';
FLUSH PRIVILEGES;
```

LP SQL contains the commands the DBMS administrator imports into the SEED web application database. Importing the commands into the database from LP SQL, the DBMS partially moved the web application from non-least privilege to least privilege. The illustration and analysis that the application has moved to least privilege is explained in Chapter 9.

7.3 Applied Case Study: Mutillidae - Automating a Least Privilege Implementation

This section describes the process of creating LP SQL from LP HERMES for the case study of Mutillidae. In Figure 6.7 a new set of least privilege users were created for the Mutillidae web application. Table 7.2 illustrates the created new least privilege user, for this case study.

Policy 1010 and Policy 1011 have the same new LP user account; however, only a single account was added to LP SQL. The SQL clause `CREATE USER IF NOT EXISTS` would prevent duplicates from being added to the database. Listing 7.5 illustrates the `CREATE` SQL commands within the file LP SQL. Listing 7.6 illustrates the `GRANT` SQL commands within the file LP SQL.

The file LP SQL contains the commands for the DBMS administrator to import into the Mutillidae web application database. Importing the users from LP SQL into the database, partially moves the web application from non-least privilege to least privilege. The illustration and analysis that the application has moved to least privilege is explained in Chapter 10.

Policy Number	New LP User
Policy 1001	home.select.blogs_table
Policy 1002	login.select.blogs_table
Policy 1003	register.select.blogs_table
Policy 1004	register.insert.accounts
Policy 1005	home.select.accounts
Policy 1006	captured-data.select.captured_data
Policy 1007	show-log.select.accounts
Policy 1008	add-to-your-blog.select.blogs_table
Policy 1009	view-someones-blog.select.accounts
Policy 1010	view-someones-blog.select.blogs_table
Policy 1011	view-someones-blog.select.blogs_table

Table 7.2: Applied Muttilidae LP SQL Users: A manually extracted list, from the HERMES file, of the policy numbers and new users that were added to LP SQL. The LP SQL file is imported by the DBMS administrator as a partial solution to move the web application from non-least privilege to least privilege.

Listing 7.5: SEED LP SQL: The SQL CREATE commands that are automatically generated from the correct and complete LP HERMES for the Mutillidae web application. The SQL file is named db-mutillidae-hpol.sql.

```
CREATE USER IF NOT EXISTS 'login.select.blogs_table'@'localhost'
    IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'register.select.blogs_table'@'localhost'
    IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'register.insert.accounts'@'localhost'
    IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'home.select.accounts'@'localhost'
    IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'captured-data.select.captured_data'@'
    localhost' IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'show-log.select.accounts'@'localhost'
    IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'add-to-your-blog.select.blogs_table'@'
    localhost' IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'view-someones-blog.select.accounts'@'
    localhost' IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'view-someones-blog.select.blogs_table'@'
    localhost' IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'view-someones-blog.select.blogs_table'@'
    localhost' IDENTIFIED BY 'passwd';
```

Listing 7.6: SEED LP SQL: The SQL GRANT commands that are automatically generated from the correct and complete LP HERMES for the Mutillidae web application. The SQL file is named db-mutillidae-hpol.sql.

```sql
GRANT SELECT on nowasp.blogs_table to 'login.select.blogs_table'@'localhost';
GRANT SELECT on nowasp.blogs_table to 'register.select.blogs_table'@'localhost';
GRANT INSERT on nowasp.accounts to 'register.insert.accounts'@'localhost';
GRANT SELECT on nowasp.accounts to 'home.select.accounts'@'localhost';
GRANT SELECT on nowasp.captured_data to 'captured-data.select.captured_data'@'localhost';
GRANT SELECT on nowasp.accounts to 'show-log.select.accounts'@'localhost';
GRANT SELECT on nowasp.blogs_table to 'add-to-your-blog.select.blogs_table'@'localhost';
GRANT SELECT on nowasp.accounts to 'view-someones-blog.select.accounts'@'localhost';
GRANT SELECT on nowasp.blogs_table to 'view-someones-blog.select.blogs_table'@'localhost';
GRANT SELECT on nowasp.blogs_table to 'view-someones-blog.select.blogs_table'@'localhost';
FLUSH PRIVILEGES;
```

Chapter 8: Systematic Process for Refactoring PHP Code to Implement Least Privilege

With a robust set toolset, LP HERMES and LP SQL, the next step in the process is to refactor the PHP code to fully the transform the web application from a non-least privilege to a least privilege model. This chapter explains the general process of manually refactoring the PHP code. Figure 8.1 illustrates **Contribution 6: A developed systematic process for PHP code modification to assist the web developer in applying least privilege permissions for securing web applications.** Furthermore, the practical application of refactoring the PHP code is applied in two case studies, the SEED Labs web application in Chapter 9 and the Mutillidae web application in Chapter 10.

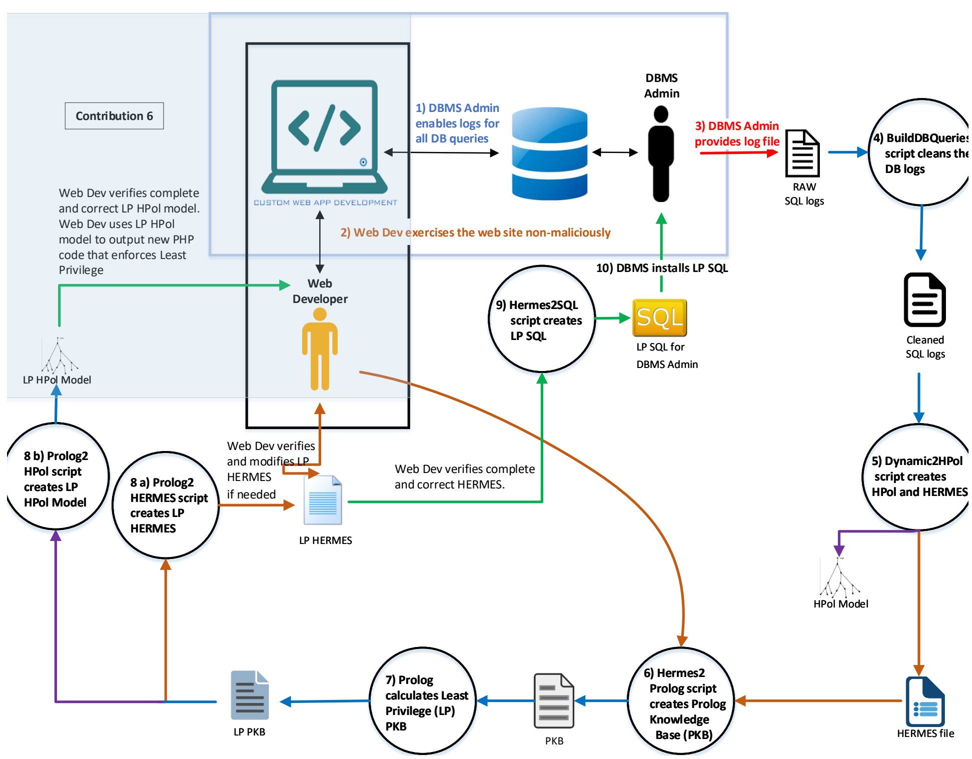


Figure 8.1: **Contribution 6: A developed systematic process for PHP code modification to assist the web developer in applying least privilege permissions for securing web applications.**

8.1 Systematic Step-by-Step for Refactoring PHP Code to Secure the Web Application

Currently, most online sites with code examples recommend using the same non-least privilege authentication pattern. This non-least privilege *root* user needs to be replaced with a least privilege user, for each query to the database. Since every web application is unique, this section explains the general process of refactoring the PHP code to move the web application to least privilege. The following general steps are:

Step 1: Identify the PHP file that holds the credentials for database connectivity. Listing 8.1, illustrates the common non-least privilege credentials.

Step 2: Create a PHP include file that contains a function to determine the appropriate user. This function will need to be passed the referrer web page. Listing 8.2 illustrates a simple function to determine the user.

Step 3: Locate all *include filename.php* that call the root configuration. These files will need to be commented out, and the determine user function will need to be referenced.

Step 4: The executeQuery function call shown in Figure 8.2 is now passed the least privilege user.

Step 5: The least privilege user makes the connection to the database and then the query is executed by the least privileged user instead of being executed by the non-least privileged user `root`.

These general step-by-step instructions should be applicable to most web applications; however, the process of refactoring the code is entirely dependent on the original web application. In Chapter 12, Section 12.5 automating this general process is discussed. The following two chapters will illustrate the refactoring process as part of two applied case studies.

Listing 8.1: Unsecured Web Application DB Example: This database configuration illustrates a non-least privilege generic root user that is common for database interaction.

```
    /* ---------------------------------------------
     * DATABASE HOST
     * ---------------------------------------------
    static public $mMySQLDatabaseHost = "127.0.0.1";

    /* ---------------------------------------------
     * DATABASE USER NAME
     * ---------------------------------------------
    static public $mMySQLDatabaseUsername = "root";

    /* ---------------------------------------------
     * DATABASE PASSWORD
     * ---------------------------------------------
    static public $mMySQLDatabasePassword = "";
```

Listing 8.2: New PHP File: This new PHP file determines the correct least privilege.

```
<?php
    function determineUser($refPage)
    {
        if($refPage == "login.php") return loginSelectAccounts;
    }
?>
```

```
public function authenticateAccount($pUsername, $pPassword, $pFromPage){

    $lQueryString =
        "SELECT username ".
        "FROM accounts ".
        "WHERE username='".$pUsername."' ".
        "AND password='".$pPassword."';";

    $pLPUser = $this->mMySQLHandler->determineUser($pFromPage);

    $lQueryResult = $this->mMySQLHandler->executeQuery($lQueryString, $pLPUser);

    if (isset($lQueryResult->num_rows)){
        return ($lQueryResult->num_rows > 0);
    }else{
        return FALSE;
    }// end if

}//end public function
```

Figure 8.2: Secured Web Application DB Example: This figure illustrates the lookup of the least privilege user. This user will execute the database query instead of the root user.

Chapter 9: Applied Case Study: SEED - Systematic Process for Refactoring PHP Code to Implement Least Privilege

This chapter describes the process of refactoring the PHP code for the case study of the SEED Labs web application. The SEED Labs is open source, and licensed under the GNU General Public License v3.0; however, I did not want to publish their labs as part of this dissertation so I modified the code and the examples. The SEED Labs manual discussed `multi_query` as an option. I chose to turn on `multi_query` for this case study. This allowed me to emulate one of their attacks as well as create my own attack.

It is important to understand the flow of the SEED web application. The first page loaded is *unsafe_home.php*. After the user enters their credentials and the login button is pressed then *unsafe_home.php*, shown in Figure 9.1, calls the page *unsafe_edit_frontend.php*, shown in Figure 9.2. The page *unsafe_edit_frontend.php* does not display a web page, instead it calls the page *unsafe_edit_backend.php* when either the *Edit Profile* button or the *Save* button is clicked. The *unsafe_edit_backend.php* page updates the database and then calls *unsafe_edit_frontend.php* until the *logout* button is clicked.

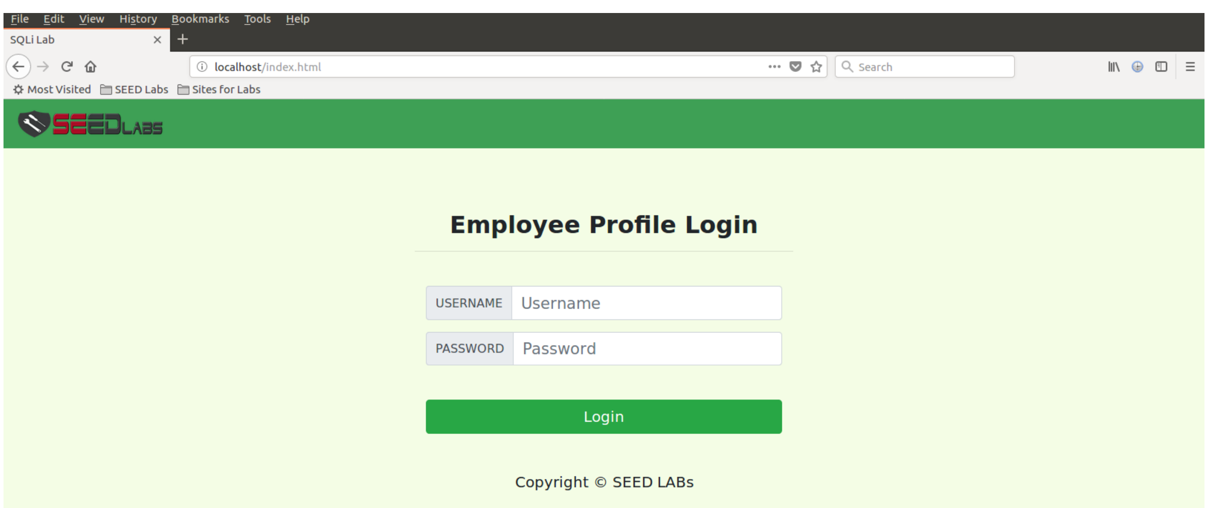


Figure 9.1: The page *unsafe_home.php*.

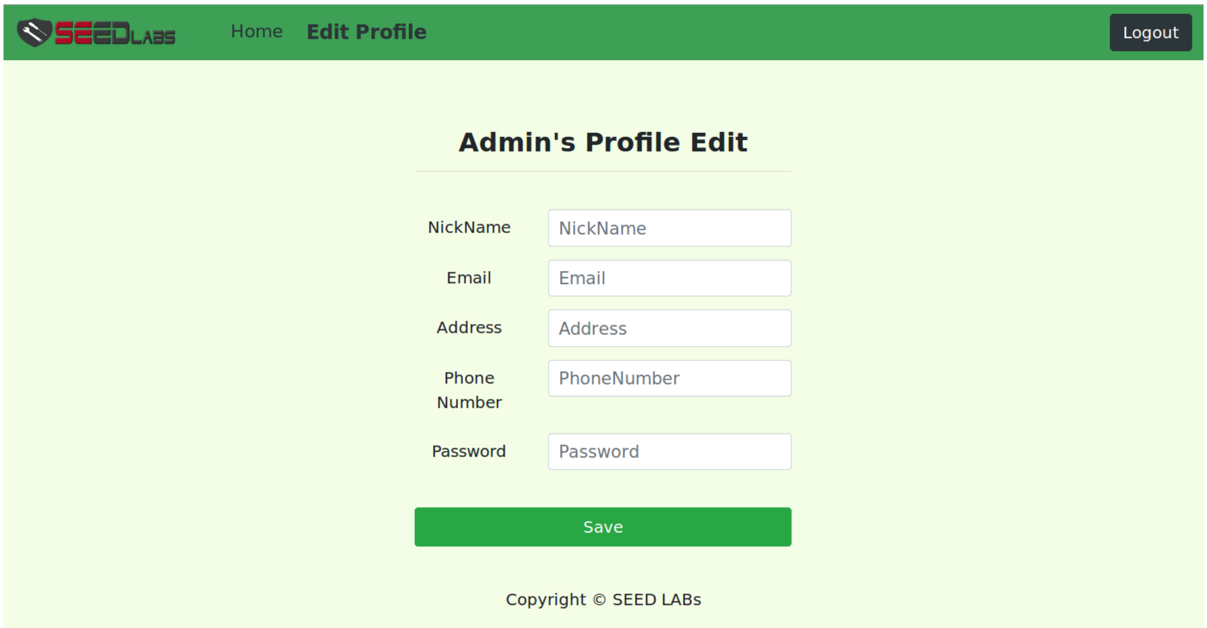


Figure 9.2: The page *unsafe_edit_frontend.php*.

Also recall that a new set of least privilege users were created for the SEED Labs web application as shown in Chapter 7 - Section 7.2. Due the limitation of the SEED Labs `mysql` database `user` table allowing a maximum of 20 characters, the new users that were created for the SEED Labs web application are `home.select.credential`, `frontend.select.credential`, and `backend.update.credential`. Furthermore, the SEED web application embeds the database credentials in each page. Listing 9.1 illustrates the least privileged SQL users credentials, which are slightly different than the credentials from Section 7.2 because of the character limitation. Figure 9.3 illustrates the `mysql.user` table after creation of the new users. The new users have no permissions on the `mysql.user` table, while Figure 9.4 illustrates that the new users have permissions only on the SEED `Users` database and `credentials` table.

Listing 9.1: SEED Users: The SQL commands creating new database users.

```
CREATE USER IF NOT EXISTS 'home.select.credential'@'localhost'
    IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'frontend.select.credential'@'localhost'
    IDENTIFIED BY 'passwd';
CREATE USER IF NOT EXISTS 'backend.update.credential'@'localhost'
    IDENTIFIED BY 'passwd';
GRANT SELECT on Users.credential to 'home.select.credential'@'
    localhost';
GRANT SELECT on Users.credential to 'frontend.select.credential'@'
    localhost';
GRANT UPDATE on Users.credential to 'backend.update.credential'@'
    localhost';
```

Host	User	Select_priv	Insert_priv	Update_priv	Delete_priv	Create_priv	Drop_priv	Reload_priv	Shutdc
localhost	root	Y	Y	Y	Y	Y	Y	Y	Y
localhost	mysql.session	N	N	N	N	N	N	N	N
localhost	mysql.sys	N	N	N	N	N	N	N	N
localhost	debian-sys-maint	Y	Y	Y	Y	Y	Y	Y	Y
%	elgg_admin	N	N	N	N	N	N	N	N
localhost	elgg_admin	N	N	N	N	N	N	N	N
localhost	phpmyadmin	N	Y	N	N	N	N	N	N
localhost	home.select.credential	N	N	N	N	N	N	N	N
localhost	backend.update.credential	N	N	N	N	N	N	N	N
localhost	frontend.select.credential	N	N	N	N	N	N	N	N

Figure 9.3: SEED Least Privilege Users: The `mysql.user` table illustrating that the new SEED users have no privileges on the DBMS middleware.

Host	Db	User	Table_name	Grantor	Timestamp	Table_priv	Column_priv
localhost	mysql	mysql.session	user	boot@connecting host	0000-00-00 00:00:00	Select	
localhost	sys	mysql.sys	sys_config	root@localhost	2017-07-25 19:46:50	Select	
localhost	Users	home.select.credential	credential	root@localhost	0000-00-00 00:00:00	Select	
localhost	Users	frontend.select.credential	credential	root@localhost	0000-00-00 00:00:00	Select	
localhost	Users	backend.update.credential	credential	root@localhost	0000-00-00 00:00:00	Update	

Figure 9.4: SEED Least Privilege Users: The SEED `Users.credential` table illustrating the new SEED users with the appropriate privileges.

Once the new users are created the next step is to refactor the web application to include the new least privileged users in place of the non-least privileged user `root`. Listing 9.2 displays the new user `home.select.credential` has replaced the user `root` for the page *unsafe_home.php*, while Listing 9.3 displays the new user `frontend.select.credential` has replaced the user `root` for the page *unsafe_edit_frontend.php* and Listing 9.4 displays the new user `backend.update.credential` has replaced the user `root` for the page *unsafe_edit_backend.php*.

Listing 9.2: LP DB Credentials: The LP SQL credentials updated for unsafe_home.php.

```
        $dbhost="localhost";
        $dbuser="home.select.credential";
        $dbpass="passwd";
        $dbname="Users";
```

Listing 9.3: LP DB Credentials: The LP SQL credentials updated for unsafe_edit_frontend.php.

```
    $dbhost="localhost";
    $dbuser="frontend.select.credential";
    $dbpass="passwd";
    $dbname="Users";
```

Listing 9.4: LP DB Credentials: The LP SQL credentials updated for unsafe_edit_backend.php.

```
    $dbhost="localhost";
    $dbuser="backend.select.credential";
    $dbpass="passwd";
    $dbname="Users";
```

The SEED Labs SQLIA web application code was slightly modified, for the pages *unsafe_home.php*, *unsafe_edit_frontend.php* and *unsafe_edit_backend.php*. Each page had the code $conn→query($sql) line of code which was changed to $conn→multi_query($sql). The difference between *query* and *multi_query* is the *query* function in PHP does not allow stacked queries while the *multi_query* function allows stacked queries. Listing 9.5 illustrates the PHP *multi_query* modification for all pages.

Listing 9.5: The code modifications to the SEED Labs web application

```
      if (!$result = $conn->multi_query($sql)) {
      $result = $conn->use_result();
```

9.1 SQL Injection Attack: SELECT Command Attack: Non-Least Privilege

The following steps outline a successful attack against the original SEED Labs web application. The attack is a tautology attack, also know as `‘or 1=1 --` attack, against the `SELECT` database command. The steps of the attack and the results of the attack are outlined below.

Step 1: Open the SEED Labs web application and execute a simple tautology `‘or 1=1 --` attack. Figure 9.5 displays the tautology attack on the non-least privilege web application.

Step 2: The *unsafe_home.php* web page issued the `SELECT` database command which allowed the tautology attack to succeed.

Step 3: The results returned the first record in the database table `credential`. Figure 9.6 displays the first record of the table `credential` which contains the user `Alice`.

Step 4: Once the tautology attack was successful the malicious actor can execute a separate attack. In this instance the malicious actor can select the *Edit Profile* button at the top of the page.

Step 5: After selecting the *Edit Profile* button, the malicious actor has full privileges to the change the record of `Alice`. Note: `Alice` could be a privileged user.

Step 6: From the *Edit Profile* page, the malicious actor can modify any information for this user, including changing the user’s password. Figure 9.7 illustrates the *edit* page where the malicious actor modified the nickname for `Alice` to become `sandy`.

Step 7: The malicious actor saves the changes. Once the changes are saved, the updated profile is displayed. Figure 9.8 illustrates Alice’s nickname has been changed.

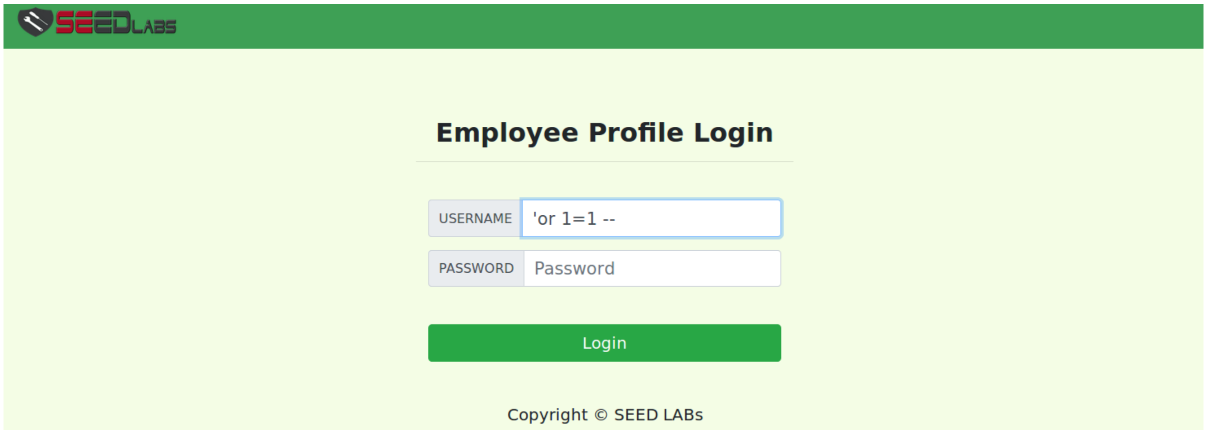


Figure 9.5: NLP SELECT Command Attack: Tautology - simple tautology attack `'or 1=1 -- ` against the `SELECT` database statement.

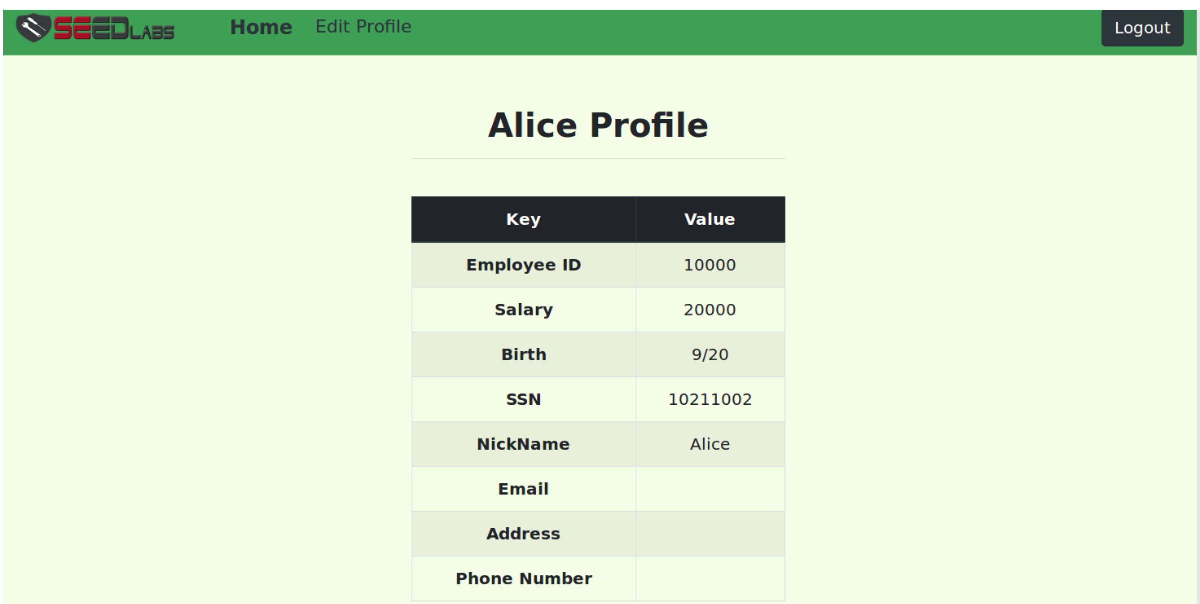


Figure 9.6: NLP SELECT Command Attack: Tautology - the tautology attack was successful. The first record of the database is displayed to the screen.

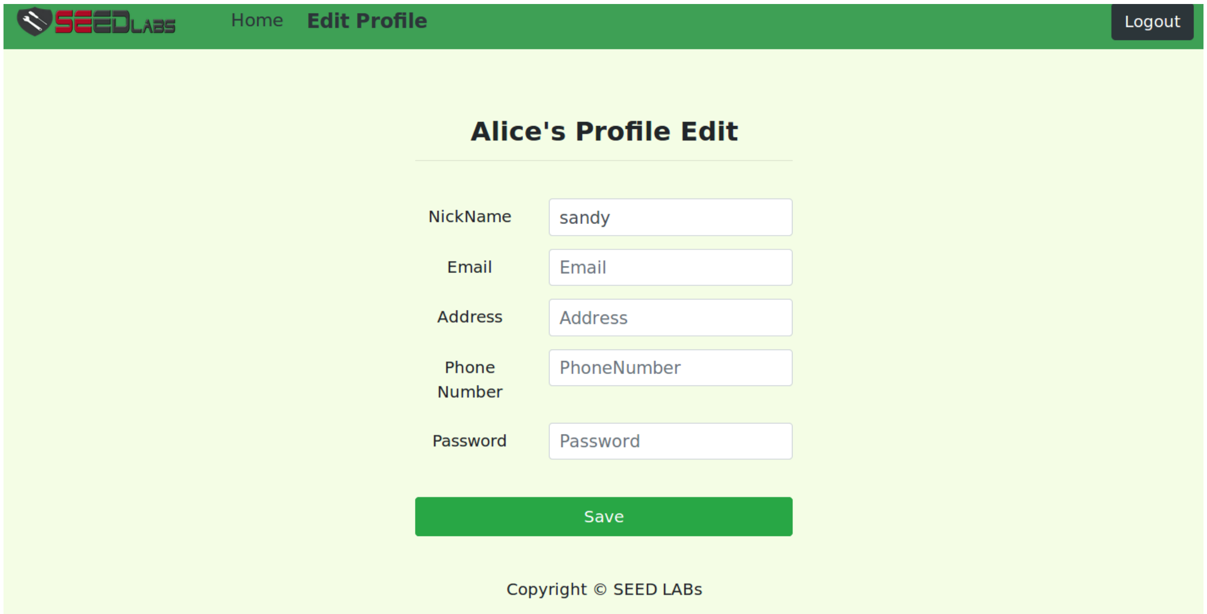


Figure 9.7: NLP SELECT Command Attack: Tautology - using the edit page *unsafe_edit_frontend.php* `Alice's` name is changed to `sandy`.

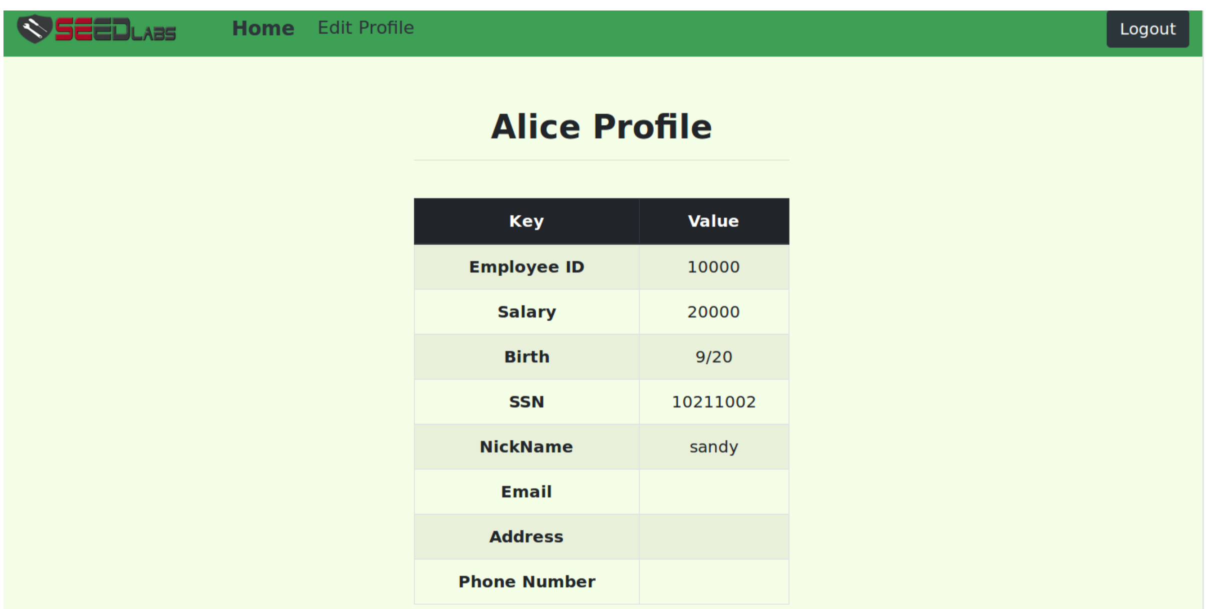


Figure 9.8: NLP SELECT Command Attack: Tautology - the results illustrating the nickname has been changed.

The tautology attack was successful. The malicious actor was able to access and modify the first record from the database table. The reason the malicious actor was able to access the first record was the tautology attack is executed on the `SELECT` statement. In this case the user `root` had full permissions to execute the `SELECT` command. Since the `UPDATE` command was being executed to modify the user's information the command succeeded because the user `root` also had `UPDATE` permissions.

9.2 SQL Injection Attack: SELECT Command Attack: Least Privilege

In this section the web application has been modified to enforce least privilege. The following steps outline a partially successful attack against the modified LP SEED web application. The attack is a tautology attack, also know as `‘or 1=1 --` attack, against the `SELECT` database command. The steps of the attack and the results of the attack are outlined below.

Step 1: Open the SEED Labs web application and execute a simple tautology `‘or 1=1 --` attack. Figure 9.9 displays the tautology attack on the least privilege web application.

Step 2: Since the *unsafe_home.php* web page is executing the `SELECT` database command, the tautology attack was successful. The results return the first record in the database table `credential`. Figure 9.10 displays the first record of the table `credential` which contains the user `Alice`.

Step 3: Once the tautology attack was successful, the malicious actor can attempt to execute a separate attack. In this instance the malicious actor can select the *Edit Profile* button at the top of the page.

Step 4: After selecting the *Edit Profile* button, the malicious actor DID NOT have privileges to change the information for Alice, since the least privilege database user is `home.select.credential`. Recall from Listing 9.1 the database users were changed.When the malicious actor attempts to modify the information for this user the command FAILS.

Step 5: From the *Edit Profile* page the malicious actor attempted to edit the nickname for the user `Alice`. Since the permissions on the web page *unsafe_edit_frontend.php* are set to `frontend.select.credential` the `UPDATE` command fails. This edit page appears to allow the malicious actor to set the nickname for `Alice`. Figure 9.11 displays the attempt to set the nickname to `Alice`. Figure 9.12 displays the information was NOT set.

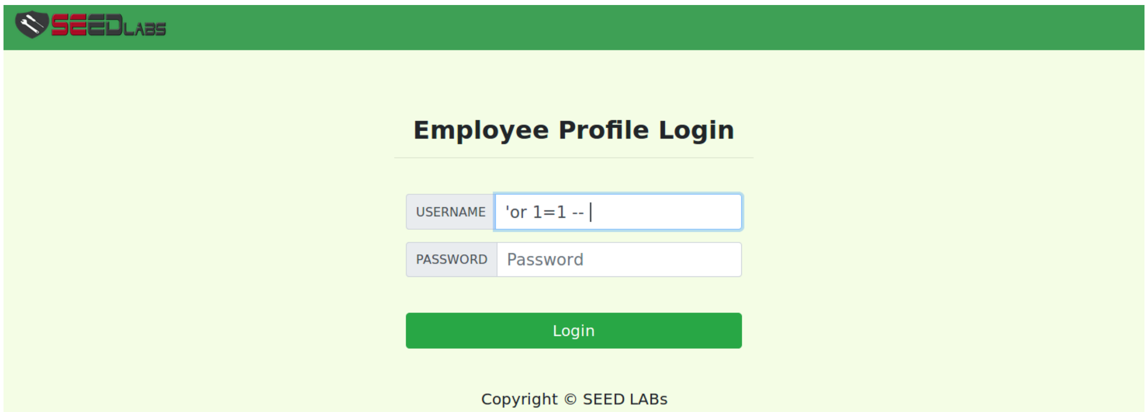


Figure 9.9: LP SELECT Command Attack: Tautology - simple tautology attack ‘`or 1=1 --` against the `SELECT` database statement.

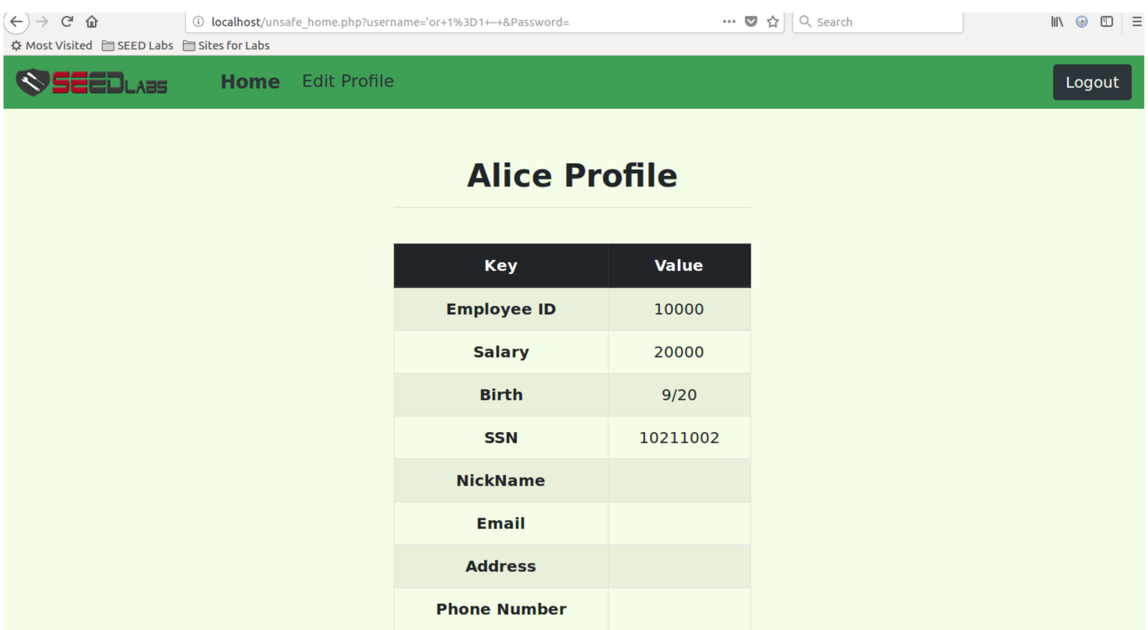


Figure 9.10: LP SELECT Command Attack: Tautology - the tautology attack was successful. The first record of the database is displayed to the screen.

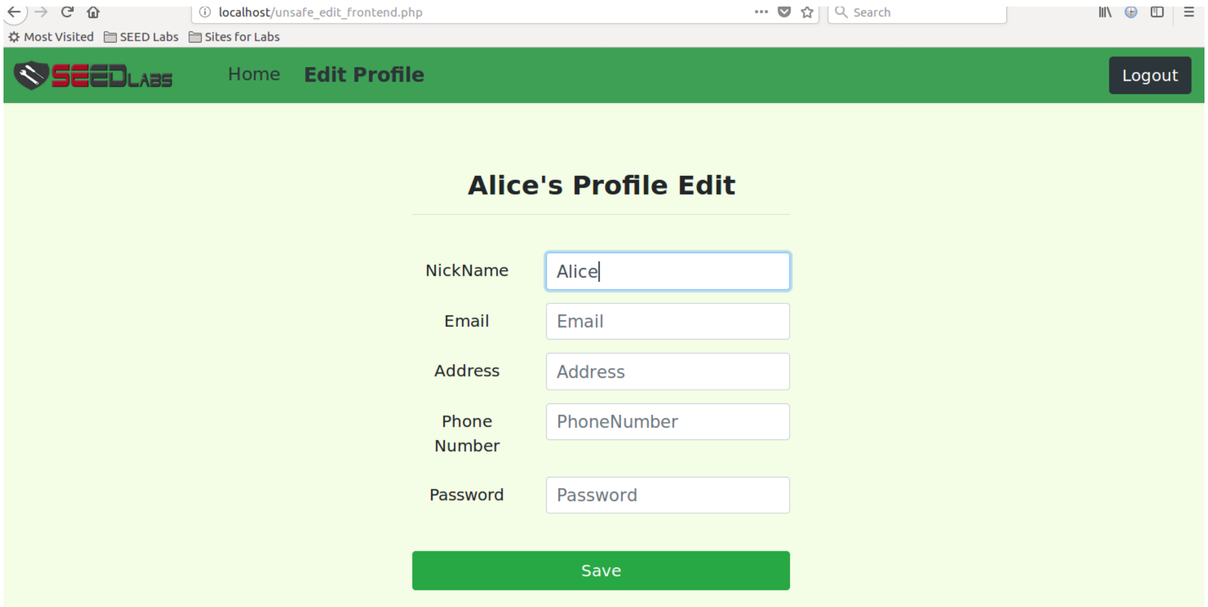


Figure 9.11: LP SELECT Command Attack: Tautology -the malicious actor attempts to add the nicknameAlice's.

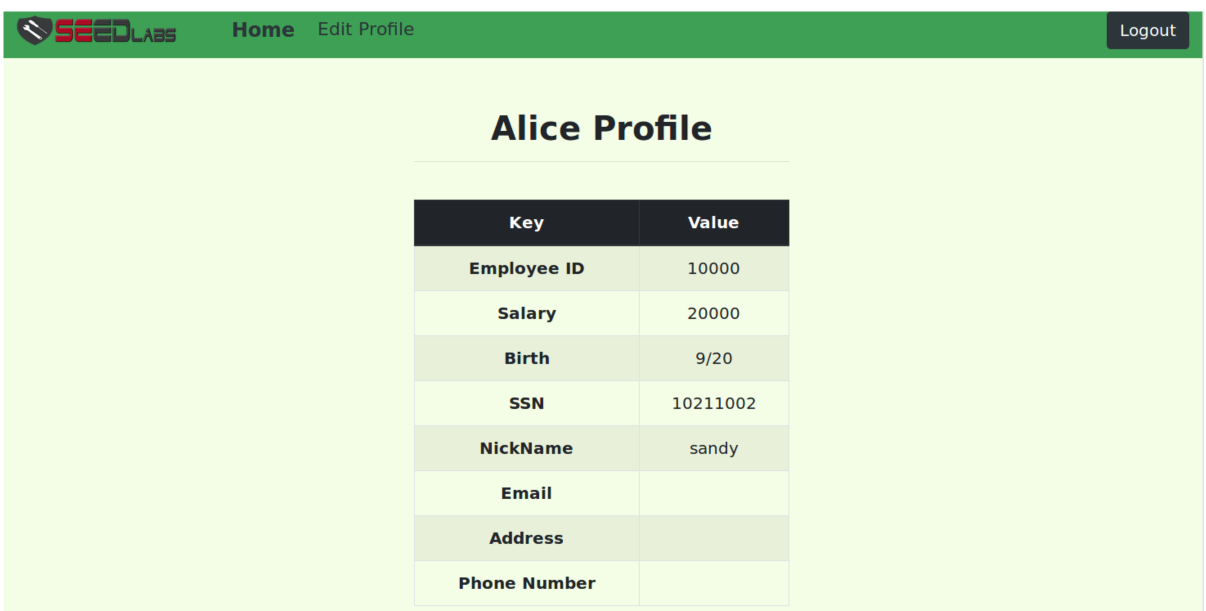


Figure 9.12: LP SELECT Command Attack: Tautology - the results illustrating the nickname has NOT been changed.

The tautology attack was successful from the `SELECT` statement being executed. The malicious actor was able to access the first record from the database table; however, attempting to change the profile for `Alice`, FAILED. Since the *unsafe_edit_frontend.php* first authenticates the user and then calls *unsafe_edit_backend.php*, eith the *multi_query* enabled and only the `SELECT` permissions, the authentication fails and the *unsafe_edit_backend.php* page is never called. The user `frontend.select.credential` only had permissions to execute the `SELECT` command not the `UPDATE` command.

9.3 SQL Injection Attack: UPDATE Command Attack Against Admin From Login Screen: Non-Least Privilege

The following steps outline a successful stacked query attack against the `admin` account of the modified SEED Labs web application. This attack is not published in the SEED Labs instructor's manual [67]. In the manual, the author discusses enabling `multi_query` to allow for attacks from the login screen. This attack is only allowed via the use of *multi_query*. The attack is considered a stacked query attack, meaning two SQL statements are executed one after the other. The attack appears as the following:

```
stu'; UPDATE credential SET Password='4b176b7bc0111ca7ba730bf6be5415f20b7b6c01'
WHERE name='admin';#
```

where '4b176b7bc0111ca7ba730bf6be5415f20b7b6c01' represents the SHA1 [68] has for the password `ownd`. The steps of the attack and the results of the attack are outlined below.

Step 1: Determine the SHA1 encoding for the password `ownd`.

Step 2: From the *unsafe_home.php* page, in the *username* field enter the stacked query attack as shown above. Figure 9.13 illustrates the attack as entered.

Step 3: Press the *Login* button.

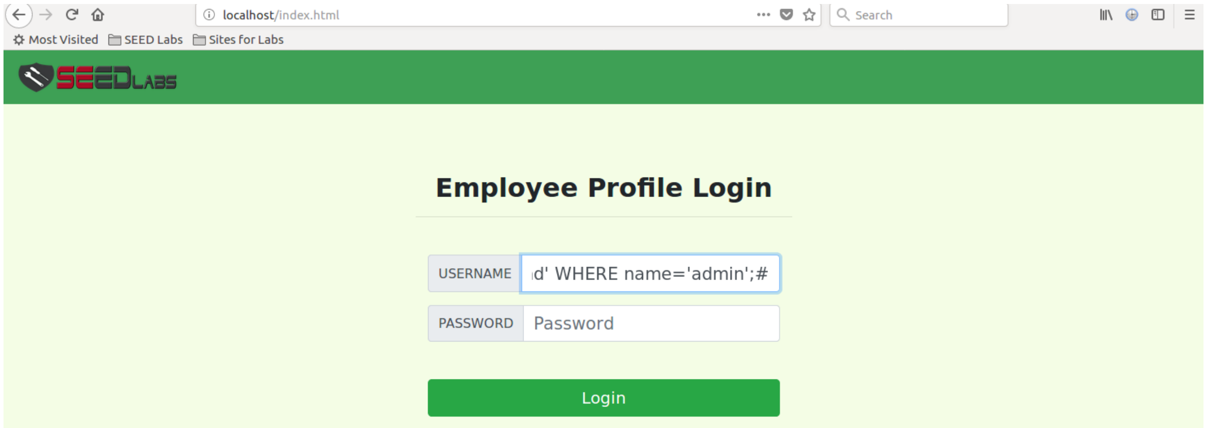


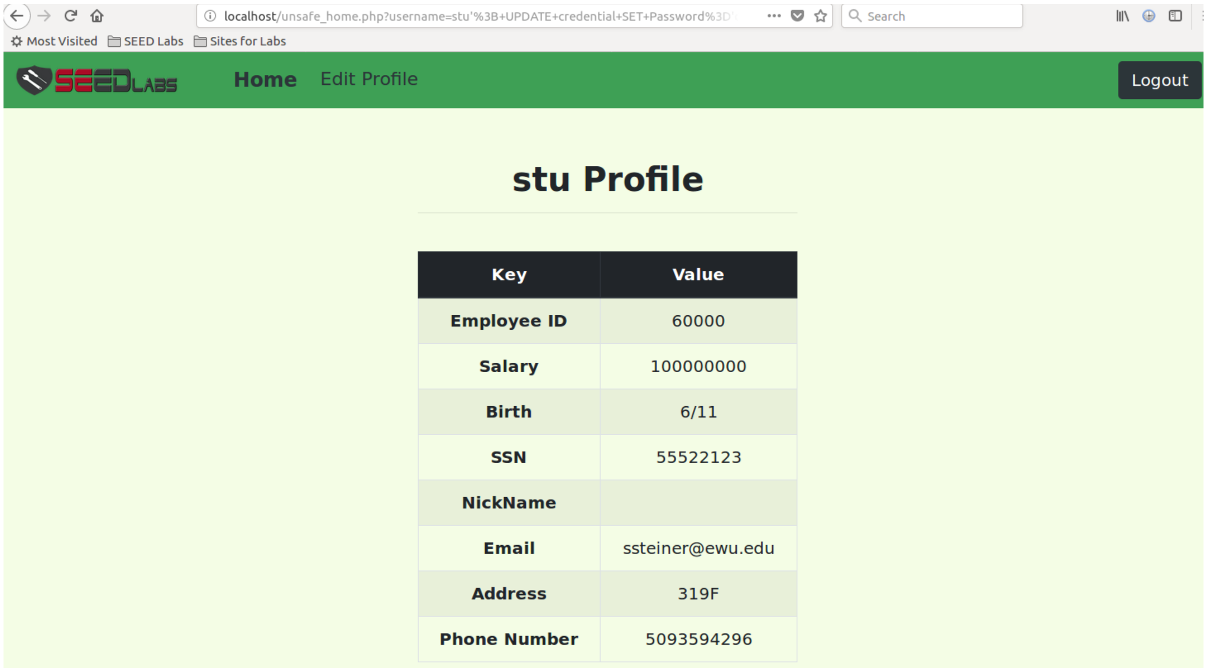
Figure 9.13: NLP UPDATE Command Attack: Login - the injection attack illustrating multiple queries in one SQL statement as entered into the *username* field.

Step 4: The *Profile* form displays the information for the user. In this case the user was `stu`. Although the user profile is displayed, the `Admin` password was modified. In this instance the password was set to `ownd`. Figure 9.14 illustrates the profile screen for the user `stu`, but behind the scenes the `Admin` password was modified.

Step 5: The malicious actor can now login into the system as the `Admin` user, utilizing the newly set password of `ownd`. Figure 9.15 illustrates that the malicious actor has logged in with `Admin` and now has full access to the database.

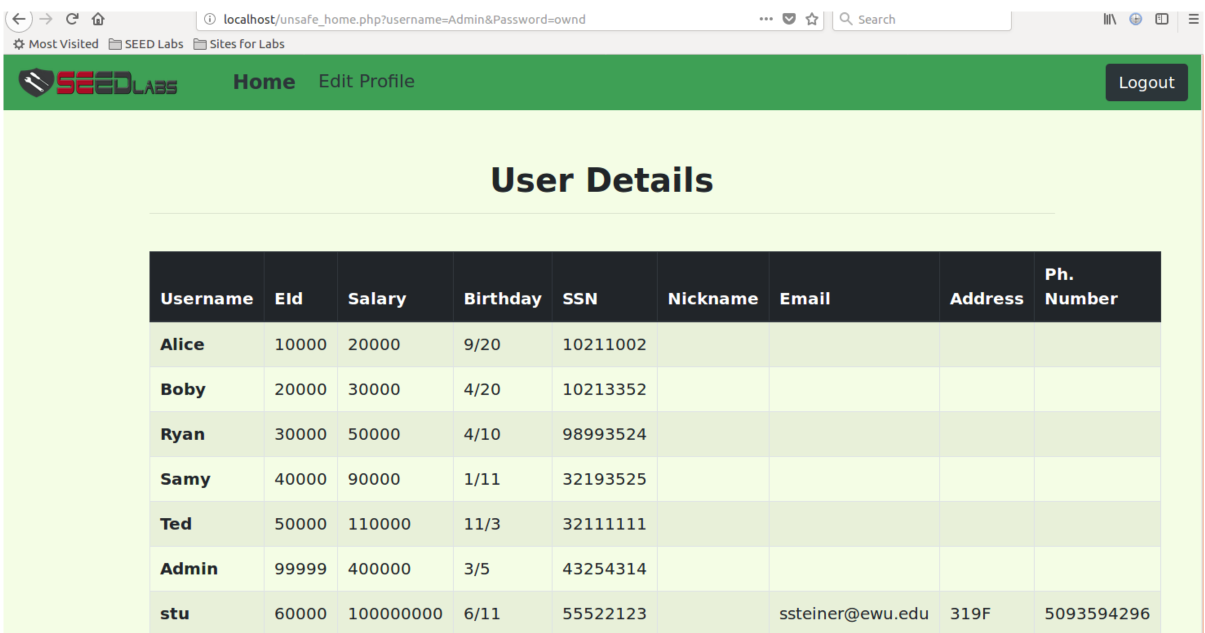
Step 6: The results after the malicious actor logged in as `Admin`. The malicious actor now has full privileges to the database.

The login attack, utilizing stacked SQL queries was fully successful. The malicious actor was able to login as a standard user; however, the malicious actor was able to modify the `Admin` password using a stacked SQL `UPDATE` command. The reason the malicious actor was able to modify the information was the `UPDATE` command was being executed by the non-least privilege `root` user.



Key	Value
Employee ID	60000
Salary	100000000
Birth	6/11
SSN	55522123
NickName	
Email	ssteiner@ewu.edu
Address	319F
Phone Number	5093594296

Figure 9.14: NLP SELECT Command Attack: Login - the results of the attack were successful; however, the only information displayed is for the user. In this case the profile is for the user `stu`.



Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				
stu	60000	100000000	6/11	55522123		ssteiner@ewu.edu	319F	5093594296

Figure 9.15: NLP UPDATE Command Attack: Login - the results after the malicious actor logged in as the `Admin` user with the password of `ownd`.

9.4 SQL Injection Attack: SELECT Command Attack Against Admin From Login Screen: Least Privilege

The SEED web application has been reset and the `Admin` password has been restored. The PHP code has been refactored for each page in the SEED web application, and the database credentials were modified to enforce the least privilege users. The following steps outline an unsuccessful attack against the `admin` account of the modified SEED Labs web application.

Step 1 - 3: Repeat the steps as shown in Section 9.3.

Step 4: The *Profile* form displays the information for the user. In this case the user was `stu`. Although the user profile is displayed the `Admin` password was **NOT** modified. In this instance the password remains set to `password`. Figure 9.16 illustrates the profile screen for the user `stu`, but behind the scenes the `Admin` password was **NOT** modified.

Step 5: The malicious actor CANNOT login into the system as the `Admin` user, utilizing the attempted set password of `ownd`.

Step 6: The results after the malicious actor FAILED to login as `Admin`. Figure 9.17 illustrates that the malicious actor COULD NOT login with `Admin`.

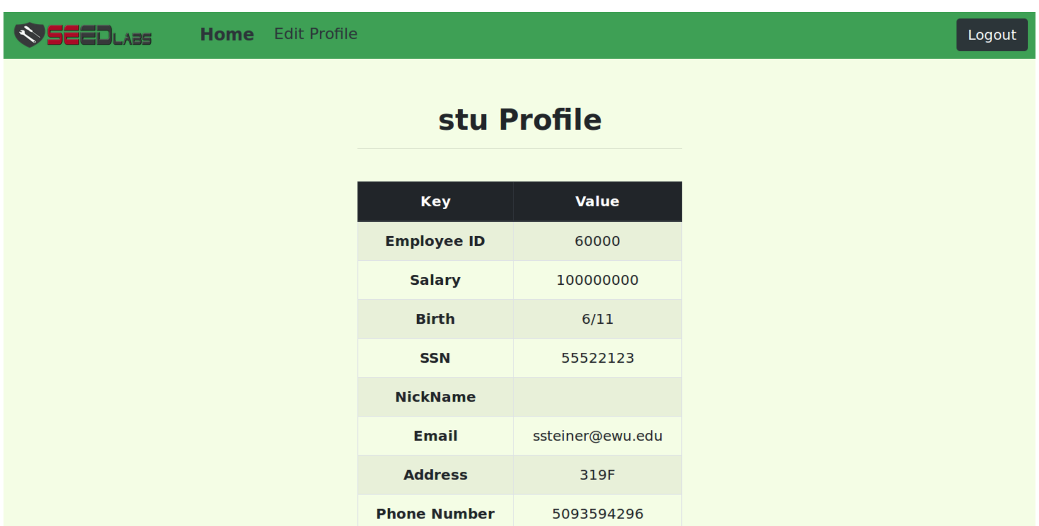


Figure 9.16: LP UPDATE Command Attack: Login - the user profile screen for `stu`, Behind the scenes the attack FAILED.

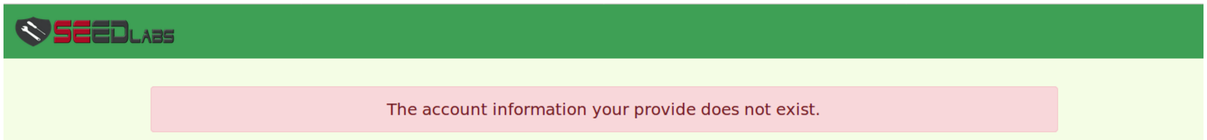


Figure 9.17: LP SELECT Command Attack: Login - the results of the attack were successful; however, the only information displayed is for the user. In this case the profile is for the user `stu`.

The login attack, utilizing stacked SQL queries was NOT successful. The malicious actor was able to login as a standard user; however, the malicious actor attempted to modify the `Admin` password using a stacked SQL `UPDATE` command. The *unsafe_home.php* first authenticates the user and then calls *unsafe_edit_frontend.php*. With the *multi_query* enabled and only the `SELECT` permissions, the authentication fails and the *unsafe_home.php* page is never called. The user `home.select.credential` only had permissions to execute the `SELECT` command not the `UPDATE` command.

9.5 SQL Injection Attack: UPDATE Command Attack Against Admin From Non-Privileged Account: Non-Least Privilege

The following steps outline a successful attack against the `admin` account of the modified SEED Labs web application, utilizing `multi_query` to allow for attacks from a non-privileged account. The concept is to issue an `UPDATE` command from a logged in general user account. The attack appears as the following:

Step 1: In the *username* field enter `stu`. In the *password* field enter `passwd`. The `stu Profile` screen displays the logged in non-privileged user.

Step 2: The non-privileged user presses the `Edit Profile` button at the top of the page.

Step 3: Determine the SHA1 encoding [68] for the password `ownd`.

Step 4: In the *NickName* field enter a command similar to the following.

`', Password= '40bd001563085fc35165329ea1ff5c5ecbdbbeef' where name='Admin' ;#`

Figure 9.18 illustrates the SQL injection that will be entered in the *NickName* field.

Step 5: Press the *Save* button.

Step 6: The *Profile* form displays the information for the user. In this case the user was `stu`. Although the user profile was displayed the `Admin` password was modified. In this instance the password was set to `ownd`. Figure 9.19 illustrates the profile screen for the user `stu`, but behind the scenes the `Admin` password was modified. Figure 9.20 illustrates the password was updated in the database.

Step 7: The malicious actor can now login into the system as the `Admin` user, utilizing the newly set password of `ownd`.

Step 8: The results after the malicious actor logged in as `Admin`. The malicious actor now has full privileges to the database. Figure 9.21 illustrates that the malicious actor could login with `Admin` and now has full access to the database.

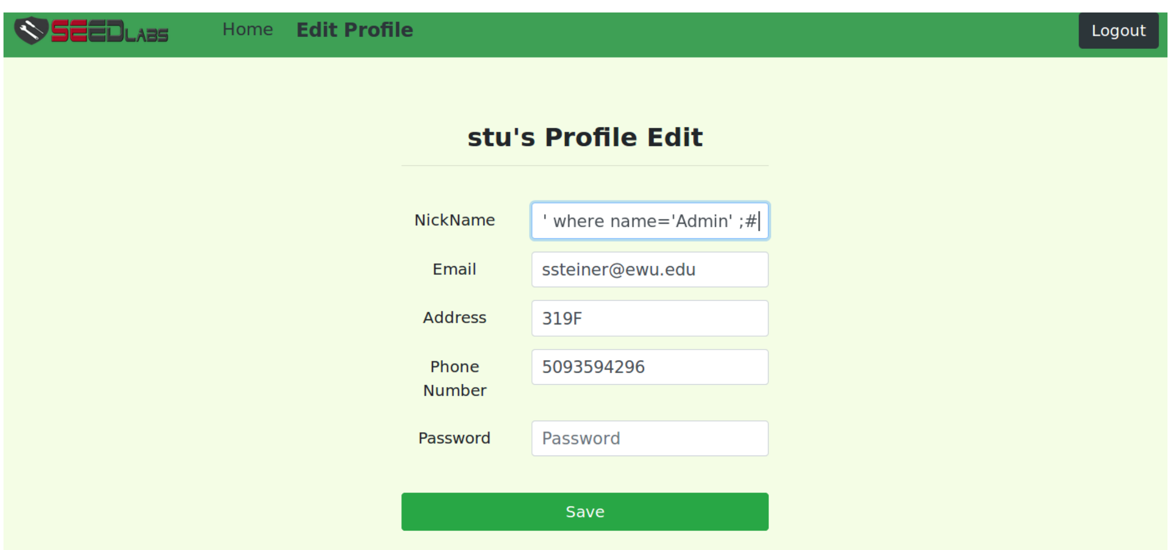


Figure 9.18: NLP UPDATE Command Attack: Non-Privileged User - the `stu's Profile Edit` screen with the injection attack entered..

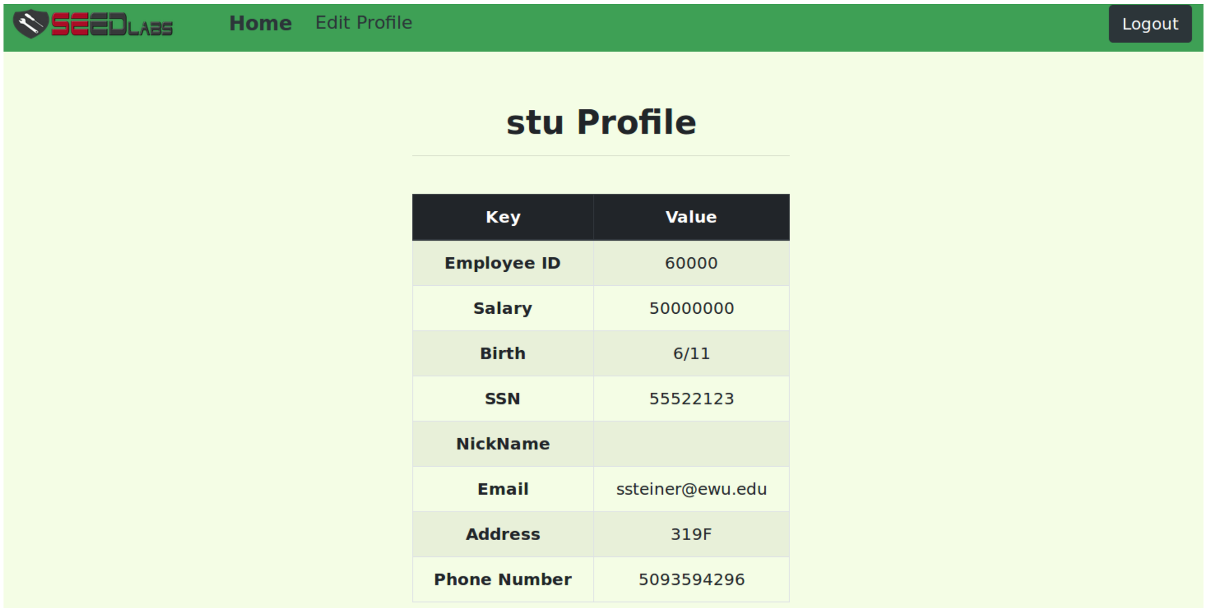
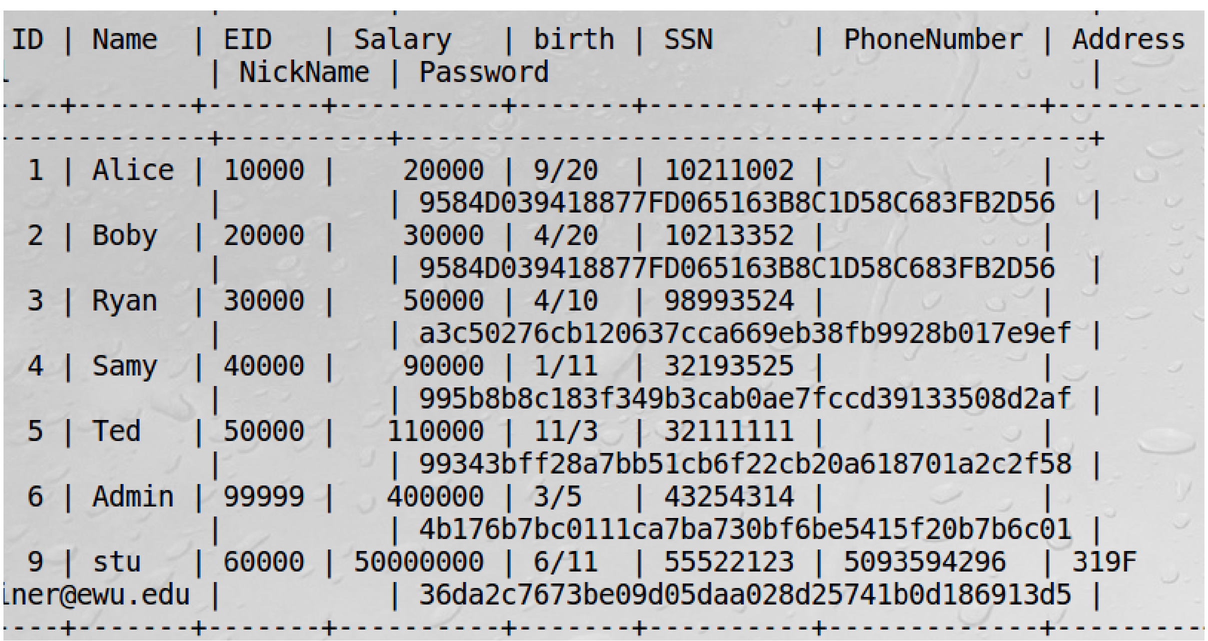
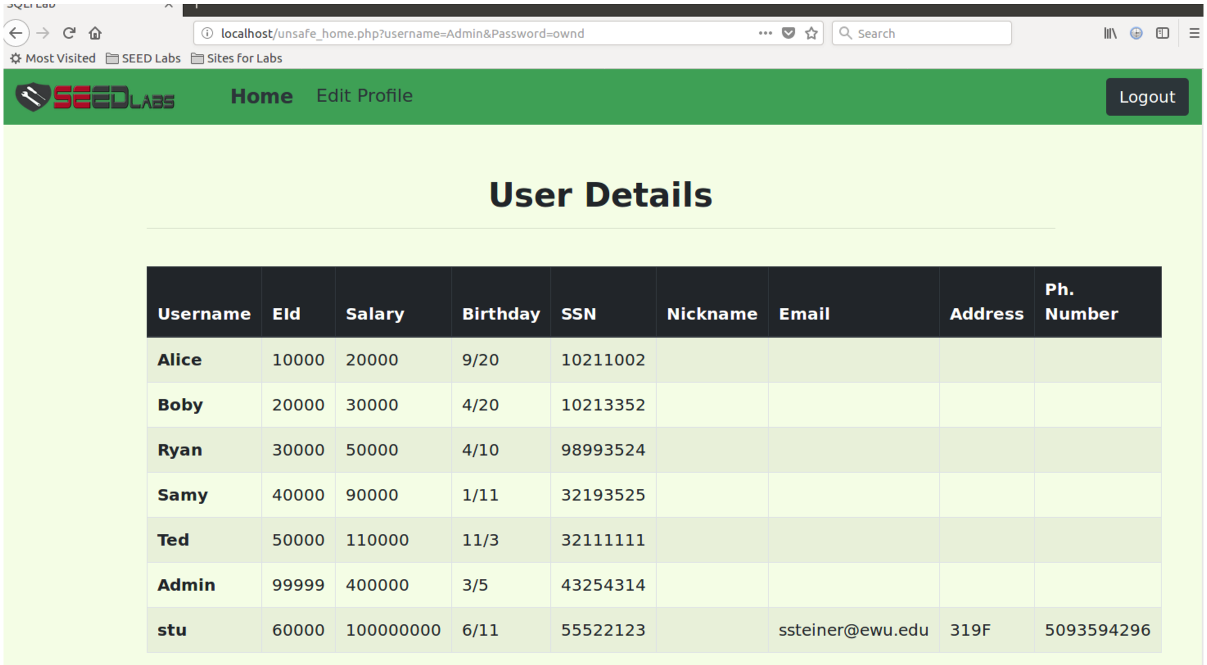


Figure 9.19: NLP UPDATE Command Attack: Non-Privileged User - the resulting profile for *stu profile* after the injection attack occurred.



ID	Name	EID	Salary	birth	SSN	PhoneNumber	Address		NickName	Password
1	Alice	10000	20000	9/20	10211002					9584D039418877FD065163B8C1D58C683FB2D56
2	Boby	20000	30000	4/20	10213352					9584D039418877FD065163B8C1D58C683FB2D56
3	Ryan	30000	50000	4/10	98993524					a3c50276cb120637cca669eb38fb9928b017e9ef
4	Samy	40000	90000	1/11	32193525					995b8b8c183f349b3cab0ae7fccd39133508d2af
5	Ted	50000	110000	11/3	32111111					99343bff28a7bb51cb6f22cb20a618701a2c2f58
6	Admin	99999	400000	3/5	43254314					4b176b7bc0111ca7ba730bf6be5415f20b7b6c01
9	stu	60000	50000000	6/11	55522123	5093594296	319F	iner@ewu.edu		36da2c7673be09d05daa028d25741b0d186913d5

Figure 9.20: NLP UPDATE Command Attack: Login - the database table of users illustrating the `Admin` password was changed.



Username	EId	Salary	Birthday	SSN	Nickname	Email	Address	Ph. Number
Alice	10000	20000	9/20	10211002				
Boby	20000	30000	4/20	10213352				
Ryan	30000	50000	4/10	98993524				
Samy	40000	90000	1/11	32193525				
Ted	50000	110000	11/3	32111111				
Admin	99999	400000	3/5	43254314				
stu	60000	100000000	6/11	55522123		ssteiner@ewu.edu	319F	5093594296

Figure 9.21: NLP UPDATE Command Attack: Non-Privileged User - The malicious actor logged in as the `Admin` with the password `ownd`.

The login attack, utilizing a non-least privileged account to issue an `UPDATE` injection query was fully successful. A non-least privileged user logged in. Once the user was logged, in the non-least privileged user became a malicious actor. The malicious actor was able to modify the `Admin` password by injecting a new password via a simple SQL `UPDATE` command. The reason the malicious actor was able to modify the information was the `UPDATE` command was being executed by the non-least privilege `root` user.

9.6 SQL Injection Attack: SELECT Command Attack Against Admin From Unprivileged Account: Least Privilege

The SEED web application has been reset and the `Admin` password has been restored. The PHP code has been refactored for each page in the SEED web application, and the database credentials were modified to enforce the least privilege users. The following steps outline a partially successful attack from a non-privileged account.

Step 1 - 5: Repeat the steps as shown in Section 9.5.

Step 6: The *Profile* form displays the information for the user. In this case the user was `stu`. Although the user profile was displayed the `Admin` password was NOT modified. In this instance the password was NOT set to `123`. Figure 9.22 illustrates the profile screen for the user `stu`.

Step 7: The malicious actor can now attempt login into the system as the `Admin` user, utilizing the newly set password of `123`.

Step 8: The results after the malicious actor logged in as `Admin`. The malicious actor now has no privileges to the database. Figure 9.23 illustrates that the malicious actor could NOT login with `Admin` and now has NO access to the database.

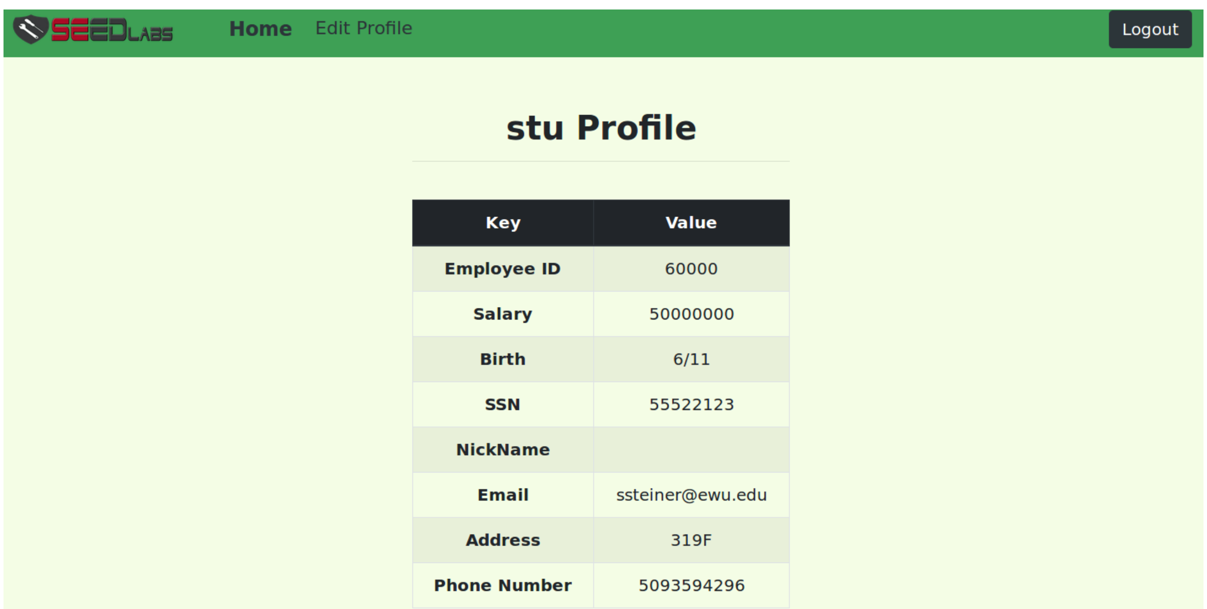


Figure 9.22: LP UPDATE Command Attack: Non-Privileged User - the resulting profile for *stu profile* after the injection attack FAILED.

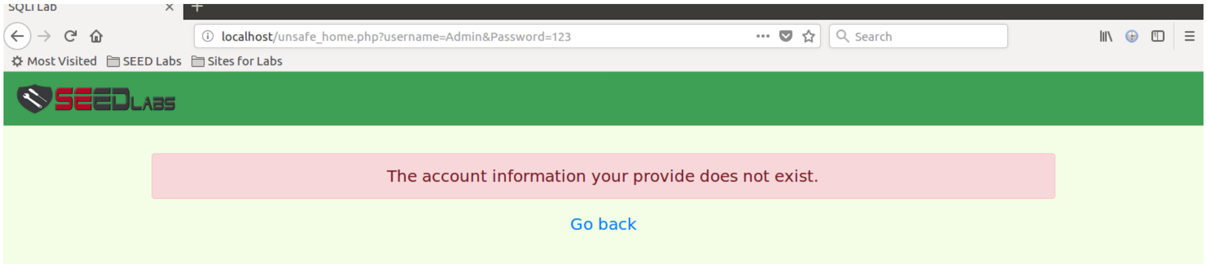


Figure 9.23: LP UPDATE Command Attack: Non-Privileged User - the results indicating the malicious actor DID NOT login as the `Admin` user with the password of `123`.

The login attack, utilizing a least privileged account to issue an `UPDATE` injection query was NOT successful. A least privileged user logged in. Once the user was logged in the non-privileged user became a malicious actor. The malicious actor was NOT able to modify the `Admin` password by injection a new password via a simple SQL `UPDATE` command. Since the `UPDATE` command was being executed to modify the user's information the command failed because the user `frontend.select.credential` did not have `UPDATE` permissions. Furthermore, using only *query* instead of *multi_query* a least privileged user that logs in a non-malicious manner, will not be able to modify their own information. This is because the page *unsafe_edit_frontend.php* is restricted to only the `SELECT` command. The ability to edit the profile is not granted because the call appears to be coming from the page *unsafe_edit_frontend.php* but in reality, the *unsafe_edit_frontend.php* is not making the `UPDATE`, it is passing the `UPDATE` information to the page *unsafe_edit_backend.php*.

9.7 SQL Injection Attack: UNION Command Attack: Non-Least Privilege

The following steps outline a successful attack against the `root` account of `mysql`. This attack utilized the original unmodified SEED web application, including the original `query` PHP command. This attack is more complicated than a basic injection attack. The steps to initiate this attack are outlined as follows.

Step 1: Open the SEED Labs web application

Step 2: From the page *unsafe_home.php*, in the *username* field enter the injection shown in Listing 9.6.

Listing 9.6: NLP: Union Attack: The injection that will provide the `MySQL` hash code of the `mysql root` password.The code modifications to the SEED Labs web application

```
' OR 1=1 UNION (SELECT 'pwned' AS id, 'MySQL-root' AS name, 0 as eid
  , 'millions' AS salary, '1900' as birth, '999-99-9999' AS ssn,
  '555-555-5555' as phonenumber, '1234 Pawn Road.' as address,
  authentication_string AS email, User AS nickname, NULL AS
  Password FROM mysql.user WHERE User='root') ORDER BY eid;#
```

Figure 9.24 illustrates the `UNION` injection typed into the *username* of the *login* form.

Step 3: Press the *Login* button.

Step 4: The results of the `UNION` injection are shown in Figure 9.25. The injection was successful and the SHA1 hash code for the `mysql root` is displayed. The resulting query of the `UNION` injection is shown in Listing 9.7.

The `UNION` attack, succeeded because the command was being executed by the non-least privilege `root` user.

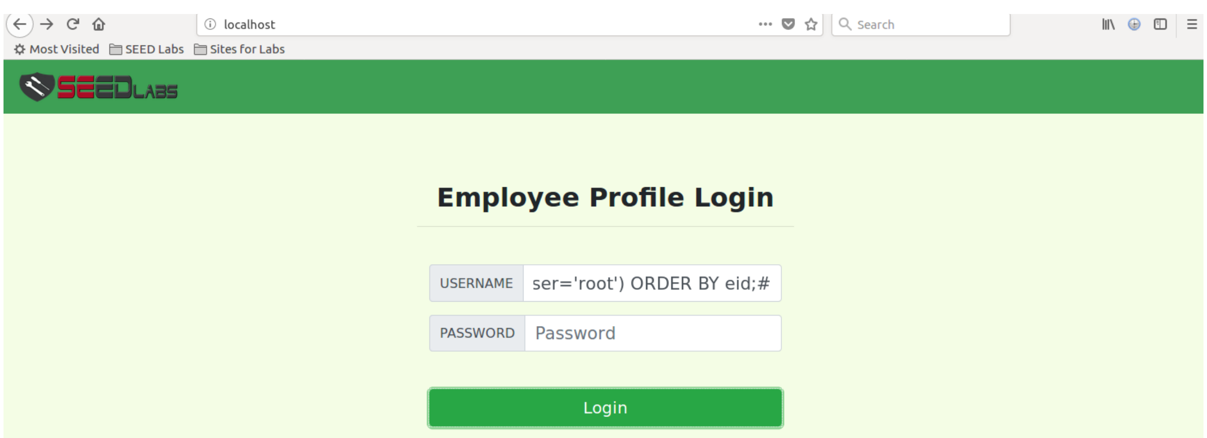


Figure 9.24: NLP UNION Command Attack: The malicious actor enters a complicated `UNION` attack as illustrated in Figure 9.6.

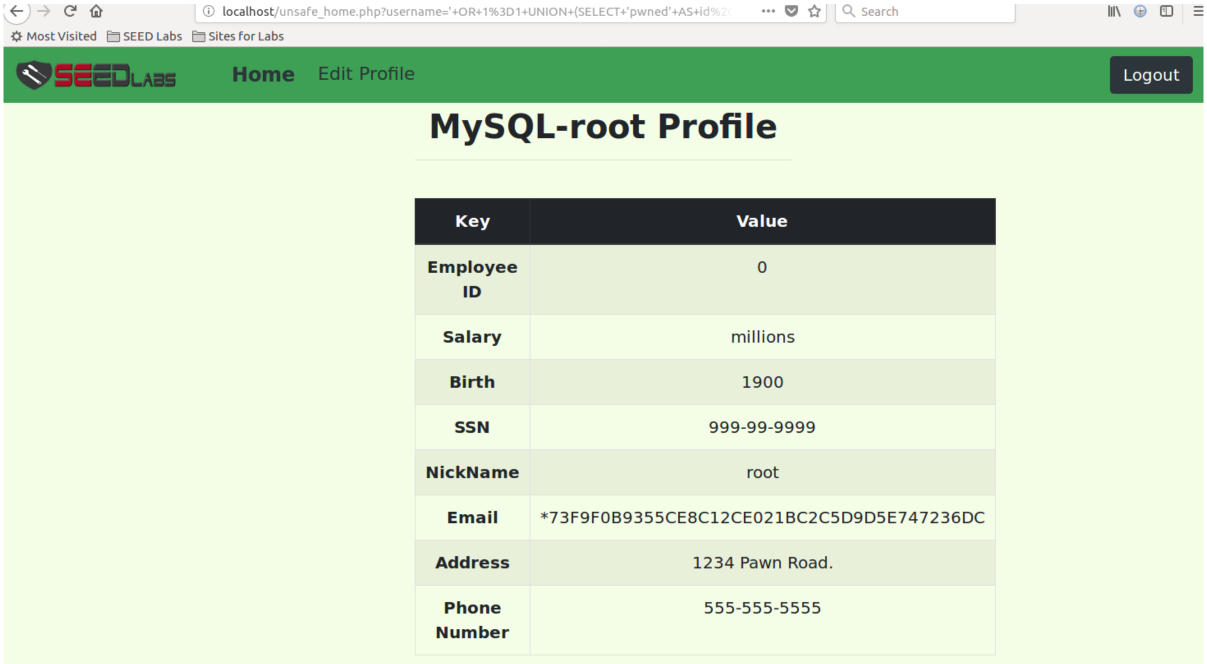


Figure 9.25: NLP UNION Command Attack: Non-Privileged User - the results after the malicious actor executed the `UNION` injection. The `mysql root` user hash code is displayed.

Listing 9.7: NLP: Union Attack: The resulting query from the UNION injection attack.

```
SELECT id, name, eid, salary, birth, ssn, phonenumber, address,
   email, nickname, Password FROM credential WHERE name='' OR 1=1
   UNION (SELECT 'pwned' AS id, 'MySQL-root' AS name, 0 as eid, '
   millions' AS salary, '1900' as birth, '999-99-9999' AS ssn,
   '555-555-5555' as phonenumber, '1234 Pawn Road.' as address,
   authentication_string AS email, User AS nickname, NULL AS
   Password FROM mysql.user WHERE User='root') ORDER BY eid;# ' and
    password='$input_pwd';
```

9.8 SQL Injection Attack: UNION Command Attack: Least Privilege

The following steps outline a FAILED attack against the `root` account of `mysql`. This attack utilized a modified version of the original SEED web application, including the original `query` PHP command. The PHP code has been refactored for each page in the SEED web application, and the database credentials were modified to enforce the least privilege users. The steps to initiate this attack are outlined as follows.

Step 1 - 3: Repeat the steps as shown in Section 9.7.

Step 4: The results of the `UNION` injection are shown in Figure 9.26. The injection was NOT successful. The reason the attack was not successful was the user `home.select.credential` only has `SELECT` on the table `credential` and not the database and table of `mysql.user`.

The `UNION` attack, FAILED because the command was being executed by the least privilege `home.select.credential` user who does not have privileges on the `mysql.user` database and table.

Figure 9.26: LP UNION Command Attack: The results after the malicious actor attempted to executed the `UNION` injection. The PHP error message is displayed. This PHP error indicates the user `home.select.credential` does not have privileges on the `mysql.user` database and table.

Chapter 10: Applied Case Study: Mutillidae - Systematic Process for PHP Code Refactoring to Implement Least Privilege

This chapter discusses the attempted process and the subsequent challenges of refactoring the PHP code for the case study of Mutillidae (2.6.42). Recall that a new set of least privilege users were created for the Mutillidae web application. The new users that were created for the Mutillidae web application are:

- `home.select.blogs_table`
- `login.select.blogs_table`
- `register.select.blogs_table`
- `register.insert.accounts`
- `home.select.accounts`
- `captured-data.select.captured_data`
- `show-log.select.accounts`
- `add-to-your-blog.select.blogs_table`
- `view-someones-blog.select.accounts`
- `view-someones-blog.select.blogs_table`

For the Mutillidae web application, the database credentials are embedded in a single page. Listing 10.1 illustrates the non-least privileged SQL credentials per page for the Mutillidae web application.

Listing 10.1: NLP SQL: The database credentials as embedded in the single PHP file for the Mutillidae web application.

```
      /* ---------------------------------------------
       * DATABASE USER NAME
       * ---------------------------------------------
       * This is the user name of the account on the database
       * which OWASP Mutillidae II will use to connect. If this is set
       * incorrectly, OWASP Mutillidae II is not going to be able to
     connect
       * to the database.
       * */
      static public $mMySQLDatabaseUsername = "root";
```

An illustration of the process to convert the web application to least privilege is as follows:

Step 1: The DBMS installed the LP SQL file into the database and created the set of new users. Figure 10.1 illustrates the DBMS has been updated with the new least privilege users.

Step 2: The `root` user was replaced by the user `home.select.accounts` Listing 10.2 illustrates the least privilege user `home.select.accounts` has replaced the non-least privileged user `root`.

Step 3: An attempt to use the web application with a simple tautology SQL injection.

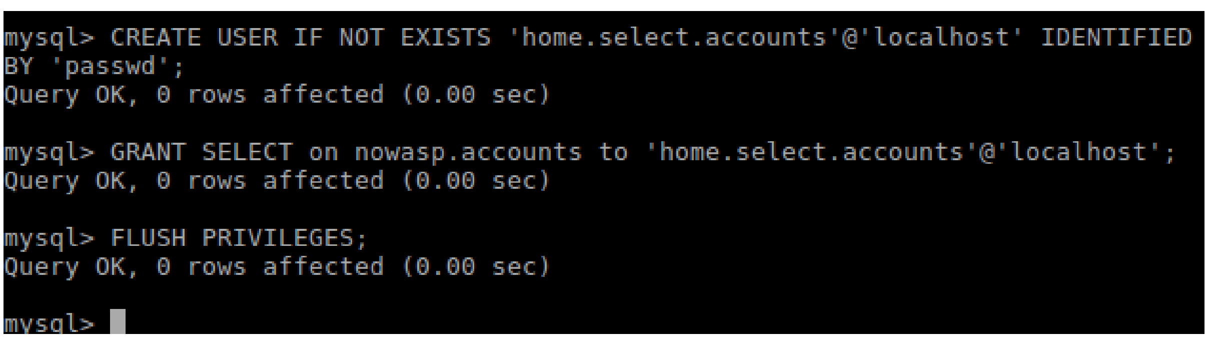


Figure 10.1: Secured Mutillidae DBMS Example: The DBMS has been updated to include the new SQL users.

Listing 10.2: LP DB Credentials: The SQL credentials have been updated in the single PHP file for the database credentials. This single file will utilize the least privilege user in place of the non-least privileged user root.

```
       * This is the user name of the account on the database
       * which OWASP Mutillidae II will use to connect. If this is set
       * incorrectly, OWASP Mutillidae II is not going to be able to
     connect
       * to the database.
       * */
      static public $mMySQLDatabaseUsername = "home.select.accounts";
```

10.1 SQL Injection Attack: SELECT Statement Attack: Non-Least Privilege

The following steps outline a successful attack against the Mutillidae web application. The attack is against the `admin` database user. The steps of the attack and the results of the attack are outlined below.

Step 1: Open the Mutillidae web application and execute a simple tautology `’or 1=1 --` attack. Figure 10.2 displays the tautology attack on the non-least privilege web application.

Step 2: The tautology attack was successful. Figure 10.3 displays a web page verifying the malicious actor is logged in as `admin`. From this page the malicious actor has access to the entire web application.

Step 3: The malicious actor has full privileges to the database. The malicious actor can modify any information within the web application.

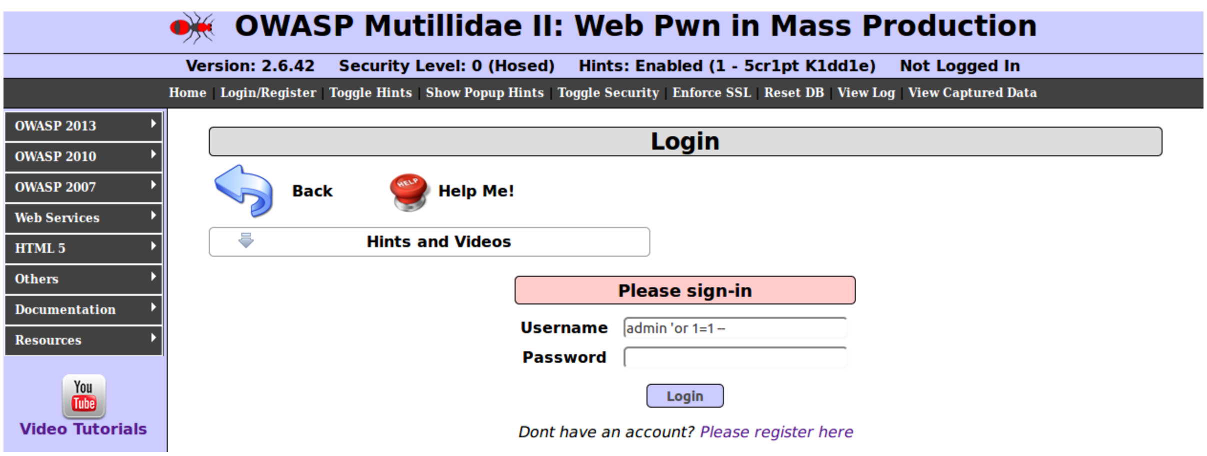


Figure 10.2: Unsecured Mutillidae Web Application Example: Simple tautology attack against the Mutillidae database admin user.

Figure 10.3: Unsecured Mutillidae Web Application Example: The tautology attack was successful. The malicious actor is logged in as `admin`.

10.2 SQL Injection Attack: SELECT Statement Attack: Least Privilege

The following are the results after refactoring the PHP code and adding the least privilege user to the Mutillidae web application. To enforce the Principle of Least Privilege the database user was modified from the non-least privilege user `root` to the least privilege user `home.select.accounts`. Once the user was updated an attempt to load the *login.php* page was performed. Figure 10.4 illustrates the results of the attempt to load the *login.php* web page for the attack from Section 10.1. The solution required granting more privileges to `home.select.accounts` then the least privilege model specifies.

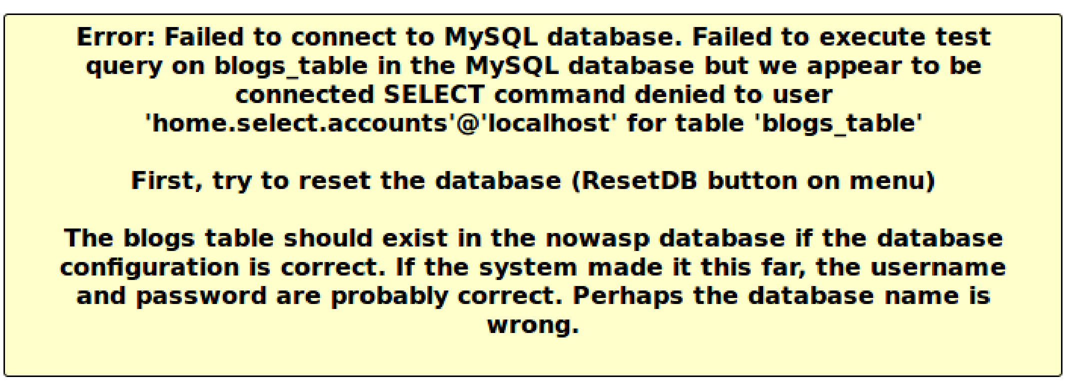


Figure 10.4: Secured Mutillidae Web Application Example: The results of exercising the Mutillidae web application regardless of the user as long as the user was not the `root` user.

Changing the users from non-least privilege to least privilege prevented any SQL injection attacks from occurring. This was true for any attack, including a tautology attack. In this case study, the least privilege formal security model was identified and implemented; however, the implement prevented attacks that were unknown. It also prevented normal execution of the web application. For the web developer to truly secure the application is currently beyond the scope of this dissertation.

Chapter 11: Related Work

This chapter discusses current existing work related to this dissertation. Existing work for securing Web applications can be categorized into Least Privilege Models discussed in Section 11.1, Secure Web Applications by Design discussed in Section 11.2, Scanning Web Applications for Security Vulnerabilities explained in Section 11.3, Reverse Engineering Web Applications discussed in Section 11.4, Dynamic and Static Mitigation Techniques discussed in Section 11.5, and Domain Specific Languages explained in Section 11.6.

11.1 Least Privilege Models

The principle of least privilege (POLP) is a well known design principle to which access control models and systems should adhere during construction or policy implementation [12, 56].

Papers concerning the least privilege model focus on applying the principle of least privilege during design, authentication, or at the operating system for hardware protection. Wang et al. [59] proposed applying the POLP at authentication time. Elliott and Knight [58, 69] proposed applying POLP during the design process. Jerbi et al. [70] proposed applying POLP to the operating system for hardware protection.

Blankenship and Freedman applied the POLP to develop Passe a replacement for the Django Web framework [57]. Passe differs from our work in that it relies on developer-supplied end-to-end test cases to learn the program flow. Our work analyzes the current Web application via our HPol security model.

The current research on the principle of least privilege differs significantly from our research, in that most of the research conducted does not address the issue of securing already developed Web applications, including addressing the issue of how to fix the current Web applications without having to completely rewrite the web.

11.2 Secure Web Application By Design

One approach for securing Web applications is the concept of building a secure Web application from the beginning. One such approach is from the company Galois.

Galois developed a secure standalone Web server with Haskell [71]. The Web applications resided separately from the Web server [72, 73]. Any new Web applications are built fresh with the concept of security built into the application.

Galois differs from our research in that our work presumes the Web application has already been built potentially without security as a focus. For example, it is difficult to determine if there are security flaws in already deployed Web applications; although, there are tools such as the burp suite [74], these tools do not illustrate all security flaws or a least privilege model.

11.3 Scanning Web Applications for Security Vulnerabilities

Scanning Web applications for security vulnerabilities is one method for identifying and correcting the existing security flaws. Fonseca et al. [75], Makino and Klyuev [76], Qianqian and Xiangjun [77], and Viera et al. [78] discussed scanning existing Web applications for vulnerabilities. Fong et al. [79] discussed building test suites for evaluation of Web application scanners. Fong et al. scanning techniques were performed against a test suite where the number of vulnerabilities was known.

Web scanners differs from our research, in that Web scanning does not have access to the original server-side application source code. Without the original source code, certain vulnerabilities cannot be identified and the POLP cannot be applied. Our research includes the original source code which allows for identification of the design vulnerabilities and better enforcement of the principle of least privilege.

Other scanners that use attack graphs are important tools for analyzing security vulnerabilities. Ou et al. [80] discusses MulVAl which uses logical attack graphs, to directly illustrate logical dependencies among attack goals and configuration information. Saha [81] discusses attack graphs by logical formulation of vulnerability analysis in an existing framework.

11.4 Reverse Engineering Web Applications

There are tools that are readily available for reverse engineering Web applications. These tools are available as browser plugins and typically only provide reverse engineering of the client-side application including the HTML, CSS, and JavaScript. Bouhissi and Malki [82], Draheim et al. [83], and Hamou-Lhadj et al. [84] report on research primarily focused on reverse engineering the client application.

Another reverse engineering technique focuses on reverse engineering the server-side applications to identify the structure of the Web application, such as the PHP code or SQL statements. Cloutier et al. [85], Guan and Yang [86], Lucca et al. [87], Tramontana [88], Tramontana et al. [89], and Weijun and Xianming [90] propose solutions that reverse engineer the Web application where the structure of the Web application is extracted and visualized as an attempt to recover the architecture of the Web application.

This research differs from the research in reverse engineering Web applications in that our research focuses on reverse engineering the Web application to create a complete security model. Furthermore, the papers on reverse engineering typically do not have access to the server-side source code, while this research assumed there was access to the server-side source code.

11.5 Survey of Dynamic and Static Mitigation Techniques Papers

Dynamic and static mitigation technique papers can be loosely grouped into three categories: papers that classify the mitigation technique to the seven SQLIA types, papers that discuss the strengths and weaknesses of each mitigation technique, including whether the technique is defensive and/or preventive, and papers that discuss the classification of SQLIAs with an analysis of the risks associated with each attack. Table 11.1 displays the classification of each survey paper. The summary of each paper is as follows:

Abirami et al. [91] provide a review of the types of SQL injection attacks as well as an analysis of several mitigation techniques.

Amirtahmasebi et al. [19] review the defense mechanisms for six mitigation techniques by discussing very specific details of the defense technique including which SQL injection type the technique protects.

Grupta et al. [20] propose a classification of the defense techniques of the static analysis based approaches. This survey paper explores eleven techniques from 2005 through 2012.

Halfond et al. [8] classify the SQL injection attack types. These attack types became the standard attack types that papers cite. In this survey paper, 17 SQLIA mitigation techniques are compared to the SQL injection attack type, including a classification of the technique as a detection or prevention technique. This paper also includes additional information about modifying the code base and additional infrastructure.

Johari and Sharma [21] present a survey of 14 prevention techniques that are either SQL injection prevention techniques or cross site scripting prevention techniques. This paper presents a description of each technique. The authors state this "... should not excuse developers from applying preventive coding techniques ... "

Junjin [26] presents an approach for SQL injection vulnerability detection; however, one half of the paper is dedicated to analyzing two other detection techniques. The analysis includes a description of manual approaches and automated approaches for prior SQL injection detection.

Kaur and Kour [22] identify and analyze the various reasons for SQL injection attacks. The paper presents the attack and an example of the attack, but does not present individual mitigation techniques.

Kindy and Pathan's [24] paper provides a detailed review of the various types of SQLIAs including an attempt to classify the individual vulnerabilities into types. These vulnerability types are mapped to the SQLIA types. This paper describes 13 mitigation techniques, including tables mapping each technique to SQL prevention or SQL detection technique, and the SQL injection attack types.

Kumar and Pateriya's [25] survey provides a review of the various types of SQLIAs including an example of each type. The 21 surveyed papers are mapped to the SQLIA types, including mapping the technique to SQL prevention or SQL detection technique, and whether, the technique generates a report.

Table 11.1: A survey and self classification of dynamic mitigation technique papers, presented in alphabetical order by author.

Authors	Mitigated SQLIA Types	Strengths & Weaknesses	Risk Analysis	Year
Amirtahmasebi et al. [19]	✓			2009
Grupta et al. [20]		✓		2014
Halfond et al. [8]	✓			2006
Johari and Sharma [21]		✓		2012
Junjin [26]		✓		2009
Kaur and Kour [22]			✓	2015
Kindy and Pathan [24]	✓			2011
Kumar and Pateriya [25]		✓		2012
Mukherjee et al. [27]		✓		2015
Sadeghian et al. [28]		✓		2013
Sajjadia and Pour [29]		✓		2013
Shar and Tan [11]		✓		2013
Sharma and Jain [30]			✓	2014
Tajpour et al. [31, 32, 33]		✓		2010

Mukherjee et al. [27] provides a review of the SQLIA problem, including the attack type and an example of each attack type. The paper reviews 17 defensive techniques with a classification of each technique as either a prevention or detection technique.

Sadeghian et al. [28] presents a review of 15 mitigation techniques. This paper classifies each mitigation as either a best coding practice technique, a detection technique, or a prevention technique.

Sajjadia and Pour [29] provides a taxonomy of prevention and detection techniques. The paper classifies the SQLIA based on the vulnerability type. The paper also addresses prevention techniques as solely static or as hybrid, both static and runtime. This review of eight techniques classifies each technique as prevention or detection, and it includes whether the code base is modified or if there is additional required information for the developer.

Shar and Tan [11] present an analysis of fifteen SQLIA defensive techniques. This paper separates each technique into the categories of defensive coding, detection techniques and runtime techniques, and it reclassifies prevention as runtime techniques and detection as static analysis techniques. This paper states that “Numerous off-the-shelf offerings are useful for quickly detecting the presence of SQLIVs [SQL injection vulnerabilities] in websites." The paper briefly mentions one runtime technique is being commercialized.

Sharma and Jain's [30] paper discusses the classification of SQL injection attacks, including the risk of each attack type. The paper also classifies the vulnerability of each SQLIA, and discusses the anatomy of orderwise injection types. This paper does not examine any specific defensive technique.

The three papers of Tajpour et al. [31, 32, 33] present the definition of SQLIAs, and the different attack types, including an example of the attack type. The papers discuss 23 mitigation techniques including mapping the technique to the seven attack types.

11.6 Domain-Specific Languages

A computer language specialized to a particular application domain is known as a domain-specific language (DSL). Hypertext Markup Language (HTML) is an example of a domain-specific language. HERMES is another example of a DSL. This section discusses a few of the domain-specific languages related to this dissertation.

The majority of flaws found in software originates during the specification stage of the system requirements. The use of domain-specific languages has shown to be a valuable resource in this part of the process. Hamdi et al. [92] introduces a DSL that is a combination of a special-purpose language and a general-purpose language. The proposed DSL is meant to reuse security infrastructure for new policies while easily allowing the expression of complicated security policies. Visic et al. [93] discusses a solution to modeling the acquisition of domain knowledge and requirements, via the deployment of a usable modeling tool. Bergel et al. [94] discusses a DSL for visualizing software dependencies as graphs. The DSL and the graph visualizes the dimensions to software metrics, the composition of the graph layout, and the graph's hierarchical edges.

This chapter provided a summary of related works as related to this dissertation. The main concepts related to this dissertation were least privilege models, security by design, scanning of web applications to identify vulnerabilities, reverse engineering web applications, dynamic mitigation techniques, static mitigation techniques, and domain specific languages.

Chapter 12: Conclusion and Future Work

This chapter summarizes the work discussed in this dissertation. Section 12.1 summarizes the contributions discussed in this dissertation, Section 12.2 discusses the value of this dissertation, Section 12.6 states the conclusions of this dissertation, Section 12.3 discusses the assumptions and the limitations of this dissertation, Section 12.4 discusses the threats to the validity of the work in this dissertation, and Section 12.5 discusses avenues for future work.

12.1 Summary of the Contributions of this Dissertation

This dissertation provides a least privilege semi-automated approach to preventing cyber attacks on web applications. Furthermore, the work presented in this dissertation provides a formal, repeatable, and automated approach and associated toolset for determining and applying least privilege permissions at the database level for securing web applications. This dissertation provided:

Contribution 1 was a manual but formal, systematic and repeatable process for securing current web applications based on the principle of least privilege.

Contribution 2 was formal web application security policy modeling.

Contribution 3 was a formal High-Level Easily Reconfigurable Specification.

Contribution 4 described the approach and associated tools for automatically learning the database-level permissions needed on the database management system for a web application to operate with the least privilege possible.

Contribution 5 explained the formal, repeatable, and automated approach and associated toolset for determining and applying least privilege permissions at the database level for securing web applications.

Contribution 6 described the systematic process for PHP code modification to assist the web developer in applying least privilege permissions for securing web applications, as well as the evaluation of the system.

12.2 Value of this Dissertation

The value of the work in this dissertation is two-fold. The first value of this dissertation is derived from the existing immediate need to fix the large number of unsecure websites today. Most modern solutions attempt to mitigate the process via new techniques of string sanitization. Query sanitization only focuses on the database, and ignores the filesystem. The solution provided is an all inclusive systematic, formal and repeatable process that was created to help developers determine how to systematically fix the web application. This solution does not just focus solely on the database, this solution also helps mitigate the filesystem. This solution implements a holistic view of the database, including the SQL code, and the filesystem. Although, the process is systematic, formal and repeatable, it is still a manual process for the code. The web developer has to manually change some code in the web application. If the web developer implements this approach, with minimal code modifications to the web application source code, the web application be can be secured.

The second value of this dissertation is this is the first attempt to automate the process. In the past, the attempt to automate the process focused solely on different attempts to sanitize the input strings. No other approach describes a semi-automated approach to holistically secure the web application via least privilege. Although a portion of the process is minimally modifying the web application source code. In Section 12.5 there is a discussion to completely secure the web application by modifying the web application source code automatically.

12.3 Assumptions and Limitations of this Dissertation

In order to create a least privilege semi-automated approach to securing web applications the following assumptions and limitations were made. The limitations are clearly outlined below, otherwise the statement is an assumption.

- The web application administrator must allow access to the source code of the web application. Limitation: Without this information a baseline non-least privilege security model can't be created.
- The web application administrator must enable full database logging. The database log

must include the referrer page, the database user, and the database tables. Limitation: Without this information the semi-automated tools to dynamically create the formal non-least privilege security model, as well as the least privilege security model will fail.

- The web application administrator will need to log into the database and grant the proper permissions that enforce the principle of least privilege.

- The web application administrator will either need to change the source code or allow a third party to change the source code. The source code needs to be updated so any database call indicates the appropriate least privilege user.

- The web application must be written in PHP. Limitation: with minor changes the approach could handle other web application programming languages.

- The database for the web application must be a SQL type database. Limitation: with minor changes the approach could handle other database types than SQL.

- It is presumed the web application will be located in */var/html/www/web-application-directory*. If the application resides in a different location, the HPol hardcoded value must be changed.

- The web server should be Apache, if it is something different the HPol hardcoded value must be changed.

- Maintainability of the large number of new users is possible my using PHP include files. These PHP include files will allow for easily adding and removing database users.

In summary if any of the above limitation items are not included, then the semi-automated approach to securing the web application will not produce a least privilege model.

12.4 Threats to Validity of this Dissertation

Threats to validity of this dissertation are defined as any factors that reduce the generality of the results. Therefore, the threats to this dissertation can be defined by:

- Selection Bias - Selection bias [95] is defined as “the selection of individuals, groups or data for analysis in such a way that proper randomization is not achieved, thereby ensuring that the sample obtained is not representative of the population intended to be analyzed.” In this dissertation there could be selection bias based on the case studies. The case studies were chosen from web applications that were educational in nature, and because the web applications were susceptible to being completely compromised. This work did not include an actual running website. We queried a few web developer and DBMS administrators that were colleagues to provide the database logs; however, no logs were provided.

- Constructs and Methods Bias - Constructs and Methods bias [96] is defined as “in a research study you are likely to reach a conclusion that your program was a good operationalization of what you wanted and that your measures reflected what you wanted them to reflect.” Similar to Selection Bias, this dissertation could contain constructs and methods bias. For the same reason as selection bias, the case studies, from an academic point of view this research reached conclusions based solely on web applications that were educational in nature, and because the web applications were susceptible to being completely compromised.

In summary, there may be threats to the validity of this dissertation based on the selection of web applications that were educational in nature; however, although educational in nature, it is believed that the case study selections emulate the ’real world’ unsecured web applications.

12.5 Future Work

This dissertation proposed a systematic method and associated tool-set for protecting web applications with a minimal need to manually modify the web application source code. This tool-set is still in its infancy. In order to be a fully formal, systematic, and automated process the following future work is considered.

HPol Further Enhancements

HPol is still in its infancy, and although enhancements were made further enhancements need to be considered.

- The first enhancement should be the full extraction of HPol policies. Recall HPol nodes were removed from the HPol structure as part of this dissertation. This should also occur for HPol policies. By extracting policies the application would allow for greater flexibility.
- The second enhancement should be the full addition of HPol links. Additional links, such as wildcard links were not utilized in this dissertation; however, in past research there was such a need for HPol links. Fully adding HPol links would allow for that past work to be revisited.

HERMES Enhancements

HERMES is even more in its infancy than HPol. The data structure for HERMES is a simple non-robust data structure. The data structure for HERMES needs to be enhanced so it is similar to HPol. This will naturally occur as others continue the research of HPol and HERMES.

Full Automation

Currently, the web administrator needs to manually modify select portions of the web application. The ultimate future work would be to develop a web application scanner that would identify the SQL users and the SQL query strings. Once the users and query strings were identified, then the automated process could rewrite the code to move the web application from non-least privilege to least privilege without involvement from the web developer.

12.6 Conclusions

In conclusion, this dissertation proposed and developed a systematic method and associated tool-set for protecting web applications with a minimal need to manually modify the web application source code. The problem of securing an existing web application was illustrated in three phases. In the first phase the non-least privilege behavior of the web application is learned and modeled provided by Contribution 1: A manual but formal, systematic and repeatable process for securing current web applications based on the principle of least privilege. Contribution 2: Formal web application security policy modeling. Contribution 3: A formal High-Level Easily Reconfigurable Specification.

In the second phase the web application is automatically converted to a least privilege model based on the behavior from the first phase, provided by Contribution 4: The approach and associated tools for automatically learning the database-level permissions needed on the database management system for a web application to operate with the least privilege possible. Contribution 5: The formal, repeatable, and automated approach and associated toolset for determining and applying least privilege permissions at the database level for securing web applications.

In the third phase the web application is manually modified and the system is evaluated, provided by Contribution 6: The systematic process for PHP code modification to assist the web developer in applying least privilege permissions for securing web applications, as well as the evaluation of the system.

Bibliography

[1] W. Du. (2018) Seed labs. [Online]. Available: http://www.cis.syr.edu/~wedu/seed/

[2] OWASP, "Category:OWASP Top Ten Project," 2017. [Online]. Available: https://www.owasp.org

[3] NetCraft, "Total number of Websites," 2017. [Online]. Available: https://news.netcraft.com/archives/category/web-server-survey/

[4] J. Kravitz, L. Kessem, S. Moore, L. Wiggins, and V. Paliwal, "IBM Security IBM X-Force Threat Intelligence Report 2016," 2016, IBM Security IBM X-Force Threat Intelligence Report 2016. [Online]. Available: https://developer.ibm.com/identitydev/2016/02/24/ibm-security-ibm-x-force-threat-intelligence-report-2016/

[5] S. Murugesan, "Understanding Web 2.0," *IT Professional*, vol. 9, pp. 34–41, jul 2007.

[6] "Stack Overflow Developer Survey Results," 2018. [Online]. Available: https://insights.stackoverflow.com/survey/2018

[7] OWASP, "SQL Injection." [Online]. Available: https://www.owasp.org/index.php/SQL_Injection

[8] W. G. J. Halfond, J. Viegas, and A. Orso, "A Classification of SQL-Injection Attacks and Countermeasures," in *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Arlington, VA, USA, mar 2006.

[9] N. Seixas, J. Fonseca, M. Vieira, and H. Madeira, "Looking at Web Security Vulnerabilities from the Programming Language Perspective: A Field Study," in *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, nov 2009, pp. 129–135.

[10] D. Ray and J. Ligatti, "Defining code-injection attacks," in *Proceedings of the 39th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '12. New York, NY, USA: ACM, 2012, pp. 179–190.

[11] L. K. Shar and H. B. K. Tan, “Defeating SQL Injection,” *Computer*, vol. 46, pp. 69–77, 2013.

[12] J. H. Saltzer and M. D. Schroeder, “The protection of information in computer systems,” *Proceedings of the IEEE*, vol. 63, pp. 1278–1308, 1975.

[13] D. Conte de Leon, M. G. Brown, A. A. Jillepalli, A. Q. Stalick, and J. Alves-Foss, “High-level and formal router policy verification,” *Journal of Computing Sciences in Colleges*, vol. 33, pp. 118–128, October 2017. [Online]. Available: https://dl.acm.org/citation.cfm?id=3144605.3144631

[14] OWASP, “OWASP mutillidae 2 project,” 2018, visited: May 2018. License CC-BY-SA. [Online]. Available: https://www.owasp.org/index.php/OWASPMutillidae2Project

[15] A. Jillepalli, D. C. de Leon, S. Steiner, and F. T. Sheldon, “HERMES: A high-level policy language for high-granularity enterprise-wide secure browser configuration management,” in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, dec 2016, pp. 1–9.

[16] T. Sutton, “A Complete Idiot’s Introduction to Formal Concept Analysis for Dummies to Teach Themselves,” GitHub Speaker Deck, Tech. Rep., 2013.

[17] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*, 1st ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997.

[18] R. Puppy, “NT Web Application Vulnerabilities,” *Phrack Magazine*, pp. 1–4, dec 1998. [Online]. Available: http://phrack.org/issues/54/8.html

[19] K. Amirtahmasebi, S. R. Jalalinia, and S. Khadem, “A survey of SQL injection defense mechanisms,” in *Internet Technology and Secured Transactions, 2009. ICITST 2009. International Conference for*, 2009, pp. 1–8.

[20] M. K. Gupta, M. C. Govil, and G. Singh, “Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey,” in *Recent Advances and Innovations in Engineering (ICRAIE), 2014*, 2014, pp. 1–5.

[21] R. Johari and P. Sharma, “A Survey on Web Application Vulnerabilities (SQLIA, XSS) Exploitation and Security Engine for SQL Injection,” in *Communication Systems and Network Technologies (CSNT), 2012 International Conference on*, 2012, pp. 453–458.

[22] P. Kaur and K. P. Kour, “SQL injection: Study and augmentation,” in *Signal Processing, Computing and Control (ISPCC), 2015 International Conference on*, 2015, pp. 102–107.

[23] J. G. Kim, “Injection Attack Detection Using the Removal of SQL Query Attribute Values,” in *Information Science and Applications (ICISA), 2011 International Conference on*, 2011, pp. 1–7.

[24] D. A. Kindy and A.-S. Pathan, “A survey on SQL injection: Vulnerabilities, attacks, and prevention techniques,” in *Consumer Electronics (ISCE), 2011 IEEE 15th International Symposium on*, jun 2011, pp. 468–471.

[25] P. Kumar and R. K. Pateriya, “A survey on SQL injection attacks, detection and prevention techniques,” in *Computing Communication Networking Technologies (ICCCNT), 2012 Third International Conference on*, jul 2012, pp. 1–5.

[26] M. Junjin, “An Approach for SQL Injection Vulnerability Detection,” in *Proceedings of the 2009 Sixth International Conference on Information Technology: New Generations*, ser. ITNG ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1411–1414.

[27] S. Mukherjee, P. Sen, S. Bora, and C. Pradhan, “SQL Injection: A sample review,” in *2015 6th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, 2015, pp. 1–7.

[28] A. Sadeghian, M. Zamani, and A. A. Manaf, “A Taxonomy of SQL Injection Detection and Prevention Techniques,” in *Informatics and Creative Multimedia (ICICM), 2013 International Conference on*, sep 2013, pp. 53–56.

[29] S. M. S. Sajjadi and B. T. Pour, “Study of SQL Injection Attacks and Countermeasures,” *International Journal of Computer and Communication Engineering*, vol. 2, 2013.

[30] C. Sharma and S. C. Jain, "Analysis and classification of SQL injection vulnerabilities and attacks on web applications," in *Advances in Engineering and Technology Research (ICAETR), 2014 International Conference on*, 2014, pp. 1–6.

[31] A. Tajpour, M. Massrum, and M. Z. Heydari, "Comparison of SQL injection detection and prevention techniques," in *Education Technology and Computer (ICETC), 2010 2nd International Conference on*, vol. 5, jun 2010, pp. V5——174——V5——179.

[32] A. Tajpour and M. JorJor Zade Shooshtari, "Evaluation of SQL Injection Detection and Prevention Techniques," in *Computational Intelligence, Communication Systems and Networks (CICSyN), 2010 Second International Conference on*, jul 2010, pp. 216–221.

[33] A. Tajpour, M. Z. Heydari, M. Masrom, and S. Ibrahim, "SQL injection detection and prevention tools assessment," in *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, vol. 9, 2010, pp. 518–522.

[34] P. Bisht, P. Madhusudan, and V. N. Venkatakrishnan, "CANDID: Dynamic Candidate Evaluations for Automatic Prevention of SQL Injection Attacks," *ACM Trans. Inf. Syst. Secur.*, vol. 13, pp. 14:1–39, mar 2010.

[35] T. Pietraszek and C. V. Berghe, "Defending Against Injection Attacks Through Context-sensitive String Evaluation," in *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, ser. RAID'05. Berlin, Heidelberg: Springer-Verlag, 2006, pp. 124–145.

[36] Z. Su and G. Wassermann, "The Essence of Command Injection Attacks in Web Applications," in *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '06. New York, NY, USA: ACM, 2006, pp. 372–382.

[37] G. Buehrer, B. W. Weide, and P. A. G. Sivilotti, "Using Parse Tree Validation to Prevent SQL Injection Attacks," in *Proceedings of the 5th International Workshop on Software Engineering and Middleware*, ser. SEM '05. New York, NY, USA: ACM, 2005, pp. 106–113.

[38] A. Liu, Y. Yuan, D. Wijesekera, and A. Stavrou, "SQLProb: A Proxy-based Architecture Towards Preventing SQL Injection Attacks," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 2054–2061.

[39] S. W. Boyd and A. D. Keromytis, "SQLrand: Preventing SQL Injection Attacks," in *In Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, 2004, pp. 292–302.

[40] W. G. J. Halfond, A. Orso, and P. Manolios, "WASP: Protecting Web Applications Using Positive Tainting and Syntax-Aware Evaluation," *Software Engineering, IEEE Transactions on*, vol. 34, pp. 65–81, jan 2008.

[41] A. Sadeghian, M. Zamani, and A. A. Manaf, "SQL injection vulnerability general patch using header sanitization," in *Computer, Communications, and Control Technology (I4CT), 2014 International Conference on*, 2014, pp. 239–242.

[42] A. Pramod, A. Ghosh, A. Mohan, M. Shrivastava, and R. Shettar, "SQLI detection system for a safer web application," in *Advance Computing Conference (IACC), 2015 IEEE International*, 2015, pp. 237–240.

[43] A. Makiou, Y. Begriche, and A. Serhrouchni, "Improving Web Application Firewalls to detect advanced SQL injection attacks," in *Information Assurance and Security (IAS), 2014 10th International Conference on*, 2014, pp. 35–40.

[44] S. A. Yevtushenko, "System of data analysis "Concept Explorer". (In Russian)," in *Proceedings of the 7th National Conference on Artificial Intelligence KII-2000*, 2000, pp. 127–134.

[45] Y. Shin, L. Williams, and T. Xie, "SQLUnitGen: Test Case Generation for SQL Injection Detection," Computer Science Dept., North Carolina State University, Tech. Rep., 2006.

[46] A. Kieyzun, P. J. Guo, K. Jayaraman, and M. D. Ernst, "Automatic Creation of SQL Injection and Cross-site Scripting Attacks," in *Proceedings of the 31st International*

Conference on Software Engineering, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 199–209.

[47] X. Fu, X. Lu, B. Peltsverger, S. Chen, K. Qian, and L. Tao, "A Static Analysis Framework For Detecting SQL Injection Vulnerabilities," in *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, vol. 1, 2007, pp. 87–96.

[48] C. Gould, Z. Su, and P. Devanbu, "JDBC checker: a static analysis tool for SQL/JDBC applications," in *Proceedings. 26th International Conference on Software Engineering*, may 2004, pp. 697–698.

[49] J. Huang, B. Liang, J. Zhong, Q. Wang, and J. Cai, "Vulnerabilities static detection for Web applications with false positive suppression," in *2010 IEEE International Conference on Information Theory and Information Security*, dec 2010, pp. 574–577.

[50] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting Web application vulnerabilities," in *2006 IEEE Symposium on Security and Privacy (S P'06)*, may 2006, pp. 6 pp.–263.

[51] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: A Web Vulnerability Scanner," in *Proceedings of the 15th International Conference on World Wide Web*, ser. WWW '06. New York, NY, USA: ACM, 2006, pp. 247–256. [Online]. Available: http://doi.acm.org.ezproxy.library.ewu.edu/10.1145/1135777.1135817

[52] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS'06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267349

[53] Y. Gomaa, A. E. A. Ahmed, M. A. Mahmood, and H. Hefny, "Survey on securing a querying process by blocking SQL injection," in *2015 Third World Conference on Complex Systems (WCCS)*, nov 2015, pp. 1–7.

[54] M. K. Gupta, M. C. Govil, and G. Singh, "Static analysis approaches to detect SQL injection and cross site scripting vulnerabilities in web applications: A survey," in *International Conference on Recent Advances and Innovations in Engineering (ICRAIE-2014)*, may 2014, pp. 1–5.

[55] M. Khari, P. Sangwan, and Vaishali, "Web-application attacks: A survey," in *2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom)*, mar 2016, pp. 2187–2191.

[56] F. B. Schneider, "Least privilege and more [computer security]," *IEEE Security Privacy*, vol. 1, pp. 55–59, sep 2003.

[57] A. Blankstein and M. J. Freedman, "Automating Isolation and Least Privilege in Web Services," in *2014 IEEE Symposium on Security and Privacy*, may 2014, pp. 133–148.

[58] A. Elliott and S. Knight, "Towards Managed Role Explosion," in *Proceedings of the 2015 New Security Paradigms Workshop*, ser. NSPW '15. New York, NY, USA: ACM, 2015, pp. 100–111.

[59] H. Wang, L. Liu, and W. Tian, "An authorization model of quantitative analysis of the least privilege," in *2012 6th International Conference on New Trends in Information Science, Service Science and Data Mining (ISSDM2012)*, oct 2012, pp. 283–288.

[60] N. Chomsky, "Three models for the description of language," in *Proc. Information Theory, IRE Transactions - Volume: 2, Issue: 3*, 1956, pp. 113–124.

[61] H. Shimazu and Y. Takashima, "Multimodal definite clause grammar," in *Proc. COLING '94 Proceedings of the 15th conference on Computational linguistics - Volume 2*, Stroudsburg, PA, USA, 1994, pp. 832–836.

[62] M. Johnson, "Two ways of formalizing grammars," in *Proc. Linguistics and Philosophy - Kluwer Academic Publishers - Volume 17*, Netherlands, 1994, pp. 221–248.

[63] M. Brown, "Hierachical Formal Modeling and Verification of Router Policies with an Applied Case Study to Cisco Router Con gurations," Ph.D. dissertation, University of Idaho, 2016.

[64] A. A. Jillepalli and D. Conte de Leon, “An Architecture for a Policy-Oriented Web Browser Management System: HiFiPol: Browser,” in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, jun 2016, pp. 382–387.

[65] *The XSB System - Version 3.8.x - Volume 1 -Programmerś Manual*, http://xsb.sourceforge.net/manual1/manual1.pdf.

[66] Oracle. (2018) Oracle vm virtualbox. [Online]. Available: http://www.virtualbox.org

[67] W. Du, *SEED Labs Instructor Manual*, Syracuse University, 2018.

[68] SHA1. (2018) Sha1 and other hash functions online generator. [Online]. Available: http://www.sha1-online.com/

[69] A. Elliott and S. Knight, “Start Here: Engineering Scalable Access Control Systems,” in *Proceedings of the 21st ACM on Symposium on Access Control Models and Technologies*, ser. SACMAT ’16. New York, NY, USA: ACM, 2016, pp. 113–124. [Online]. Available: http://doi.acm.org.ezproxy.library.ewu.edu/10.1145/2914642.2914651

[70] A. Jerbi, E. Hadar, C. Gates, and D. Grebenev, “An Access Control Reference Architecture,” in *Proceedings of the 2nd ACM Workshop on Computer Security Architectures*, ser. CSAW ’08. New York, NY, USA: ACM, 2008, pp. 17–24.

[71] Galois, “Galois,” 2017. [Online]. Available: https://galois.com/

[72] D. Burke, J. Hurd, and A. Tomb. (2010) High assurance software development. [Online]. Available: http://code.galois.com/paper/2010/HighAssuranceSoftwareDevelopment.pdf

[73] J. Launchbury, “Cross-domain WebDAV Server,” in *Proceedings of the 4th ACM SIGPLAN Workshop on Commercial Users of Functional Programming*, ser. CUFP ’07. New York, NY, USA: ACM, 2007, pp. 1 – 2.

[74] P. W. Security. (2018) Burp suite. [Online]. Available: https://portswigger.net/

[75] J. Fonseca, M. Vieira, and H. Madeira, “Evaluation of Web Security Mechanisms Using Vulnerability & Attack Injection,” *IEEE Transactions on Dependable and Secure Computing*, vol. 11, pp. 440–453, sep 2014.

[76] Y. Makino and V. Klyuev, "Evaluation of web vulnerability scanners," in *2015 IEEE 8th International Conference on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications (IDAACS)*, vol. 1, sep 2015, pp. 399–402.

[77] W. Qianqian and L. Xiangjun, "Research and design on Web application vulnerability scanning service," in *2014 IEEE 5th International Conference on Software Engineering and Service Science*, jun 2014, pp. 671–674.

[78] M. Vieira, N. Antunes, and H. Madeira, "Using web security scanners to detect vulnerabilities in web services," in *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, jun 2009, pp. 566–571.

[79] E. Fong, R. Gaucher, V. Okun, P. E. Black, and E. Dalci, "Building a Test Suite for Web Application Scanners," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS 2008)*, jan 2008, p. 478.

[80] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 336–345. [Online]. Available: http://doi.acm.org.ezproxy.library.ewu.edu/10.1145/1180405.1180446

[81] D. Saha, "Extending logical attack graphs for efficient vulnerability analysis," in *Proceedings of the 15th ACM Conference on Computer and Communications Security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 63–74. [Online]. Available: http://doi.acm.org.ezproxy.library.ewu.edu/10.1145/1455770.1455780

[82] H. E. Bouhissi and M. Malki, "Reverse Engineering Existing Web Service Applications," in *2009 16th Working Conference on Reverse Engineering*, oct 2009, pp. 279–283.

[83] D. Draheim, C. Lutteroth, and G. Weber, "A Source Code Independent Reverse Engineering Tool for Dynamic Web Sites," in *Ninth European Conference on Software Maintenance and Reengineering*, mar 2005, pp. 168–177.

[84] A. Hamou-Lhadj, A. En-Nouaary, and K. Sultan, "Reverse Engineering of Web

Based Systems," in *2007 Innovations in Information Technologies (IIT)*, nov 2007, pp. 193–197.

[85] J. Cloutier, S. Kpodjedo, and G. E. Boussaidi, "WAVI: A reverse engineering tool for web applications," in *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, may 2016, pp. 1–3.

[86] H. Guan, H. Yang, and H. Hakeem, "Reverse Engineering Web Applications for Security Mechanism Enhancement," in *2014 IEEE 38th International Computer Software and Applications Conference Workshops*, jul 2014, pp. 492–497.

[87] G. A. D. Lucca, A. R. Fasolino, F. Pace, P. Tramontana, and U. D. Carlini, "WARE: a tool for the reverse engineering of Web applications," in *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, 2002, pp. 241–250.

[88] P. Tramontana, "Reverse engineering Web applications," in *21st IEEE International Conference on Software Maintenance (ICSM'05)*, sep 2005, pp. 705–708.

[89] P. Tramontana, D. Amalfitano, and A. R. Fasolino, "Reverse engineering techniques: From web applications to rich Internet applications," in *2013 15th IEEE International Symposium on Web Systems Evolution (WSE)*, sep 2013, pp. 83–86.

[90] S. Weijun, L. Shixian, and L. Xianming, "An Approach for Reverse Engineering of Web Applications," in *2008 International Symposium on Information Science and Engineering*, vol. 2, dec 2008, pp. 98–102.

[91] R. D. J. Abirami and C. Valliyammai, "A top web security vulnerability sql injection attack - survey," in *2015 Seventh International Conference on Advanced Computing (ICoAC)*, Oct 2015, pp. 1–9.

[92] H. Hamdi, M. Mosbah, and A. Bouhoula, "A domain specific language for securing distributed systems," *2007 Second International Conference on Systems and Networks Communications (ICSNC 2007)*, pp. 76–76, 2007. [Online]. Available: http://ieeexplore.ieee.org/document/4300048/

[93] N. Visic, H.-G. Fill, R. A. Buchmann, and D. Karagiannis, "A domain-specific language for modeling method definition: From requirements to grammar," *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, pp. 286–297, 2015. [Online]. Available: http://ieeexplore.ieee.org/document/7128889/

[94] A. Bergel, S. Maass, S. Ducasse, and T. Girba, "A domain-specific language for visualizing software dependencies as a graph," *2014 Second IEEE Working Conference on Software Visualization*, pp. 45–49, 2014. [Online]. Available: http://ieeexplore.ieee.org/document/6980212/

[95] Wikipedia. (2018) Selection bias. [Online]. Available: https://en.wikipedia.org/wiki/Selection_bias

[96] T. Cook and D. Campbell, *Quasi-Experimentation: Design and Analysis Issues for Field Settings.* Houghton Mifflin, 1979.

Appendix A: Complete Listing of Mutillidae SQL Log

Listing A.1: Mutillidae SQL Log: Complete listing of the non-least privilege non-malicious run of the Mutillidae web application.

```
/usr/sbin/mysqld, Version: 5.7.20-0ubuntu0.16.04.1 ((Ubuntu)). started with:
Tcp port: 3306  Unix socket: /var/run/mysqld/mysqld.sock
Time                 Id Command    Argument
2017-12-26T18:59:18.029648Z     7 Quit
2017-12-26T18:59:28.522673Z     8 Connect   root@localhost on  using Socket
2017-12-26T18:59:28.523446Z     8 Init DB   nowasp
2017-12-26T18:59:28.523505Z     8 Query SELECT 'test connection'
2017-12-26T18:59:28.523577Z     8 Query SELECT cid FROM blogs_table
2017-12-26T18:59:28.528773Z     8 Quit
2017-12-26T18:59:28.529031Z     9 Connect   root@localhost on  using Socket
2017-12-26T18:59:28.529095Z     9 Init DB   nowasp
2017-12-26T18:59:28.531037Z    10 Connect   root@localhost on  using Socket
2017-12-26T18:59:28.531098Z    10 Init DB   nowasp
2017-12-26T18:59:28.531412Z    11 Connect   root@localhost on  using Socket
2017-12-26T18:59:28.531465Z    11 Init DB   nowasp
2017-12-26T18:59:28.531780Z    12 Connect   root@localhost on  using Socket
2017-12-26T18:59:28.531833Z    12 Init DB   nowasp
2017-12-26T18:59:28.535597Z     9 Query INSERT INTO hitlog(hostname, ip, browser, referer, date) VALUES ('::1', '::1', 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko/20100101 Firefox/57.0', 'User visited: /var/www/html/mutillidae/home.php',  now() )
2017-12-26T18:59:28.538616Z    10 Quit
2017-12-26T18:59:28.539283Z    12 Quit
2017-12-26T18:59:28.539291Z     9 Quit
2017-12-26T18:59:28.539296Z    11 Quit
2017-12-26T18:59:33.663113Z    13 Connect   root@localhost on  using Socket
2017-12-26T18:59:33.663196Z    13 Init DB   nowasp
2017-12-26T18:59:33.663257Z    13 Query SELECT 'test connection'
2017-12-26T18:59:33.663336Z    13 Query SELECT cid FROM blogs_table
2017-12-26T18:59:33.663516Z    13 Quit
2017-12-26T18:59:33.663703Z    14 Connect   root@localhost on  using Socket
2017-12-26T18:59:33.663765Z    14 Init DB   nowasp
2017-12-26T18:59:33.663914Z    15 Connect   root@localhost on  using Socket
2017-12-26T18:59:33.663970Z    15 Init DB   nowasp
2017-12-26T18:59:33.664220Z    16 Connect   root@localhost on  using Socket
2017-12-26T18:59:33.664326Z    16 Init DB   nowasp
2017-12-26T18:59:33.664499Z    17 Connect   root@localhost on  using Socket
2017-12-26T18:59:33.664546Z    17 Init DB   nowasp
2017-12-26T18:59:33.675667Z    16 Query SELECT level_1_help_include_files.level_1_help_include_file_key,
                     level_1_help_include_files.level_1_help_include_file_description
```

```
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                level_1_help_include_files.level_1_help_include_file_key
            WHERE page_help.page_name = 'login.php' ORDER BY page_help.order_preference
2017-12-26T18:59:33.681025Z    14 Query INSERT INTO hitlog(hostname, ip, browser, referer, date) VALUES ('::1', '::1', 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko/20100101 Firefox/57.0', 'User visited: login.php',  now() )
2017-12-26T18:59:33.684425Z    15 Quit
2017-12-26T18:59:33.684476Z    17 Quit
2017-12-26T18:59:33.684509Z    14 Quit
2017-12-26T18:59:33.684542Z    16 Quit
2017-12-26T18:59:45.042399Z    18 Connect   root@localhost on  using Socket
2017-12-26T18:59:45.042541Z    18 Init DB   nowasp
2017-12-26T18:59:45.042603Z    18 Query SELECT 'test connection'
2017-12-26T18:59:45.042682Z    18 Query SELECT cid FROM blogs_table
2017-12-26T18:59:45.042930Z    18 Quit
2017-12-26T18:59:45.043069Z    19 Connect   root@localhost on  using Socket
2017-12-26T18:59:45.043136Z    19 Init DB   nowasp
2017-12-26T18:59:45.043271Z    20 Connect   root@localhost on  using Socket
2017-12-26T18:59:45.043314Z    20 Init DB   nowasp
2017-12-26T18:59:45.043456Z    21 Connect   root@localhost on  using Socket
2017-12-26T18:59:45.043501Z    21 Init DB   nowasp
2017-12-26T18:59:45.043656Z    22 Connect   root@localhost on  using Socket
2017-12-26T18:59:45.043719Z    22 Init DB   nowasp
2017-12-26T18:59:45.052658Z    21 Query SELECT level_1_help_include_files.level_1_help_include_file_key,
                    level_1_help_include_files.level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                level_1_help_include_files.level_1_help_include_file_key
            WHERE page_help.page_name = 'register.php' ORDER BY page_help.order_preference
2017-12-26T18:59:45.053602Z    19 Query INSERT INTO hitlog(hostname, ip, browser, referer, date) VALUES ('::1', '::1', 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko/20100101 Firefox/57.0', 'User visited: register.php',  now() )
2017-12-26T18:59:45.054585Z    22 Quit
2017-12-26T18:59:45.054590Z    20 Quit
2017-12-26T18:59:45.054683Z    19 Quit
2017-12-26T18:59:45.054712Z    21 Quit
2017-12-26T19:00:08.913937Z    23 Connect   root@localhost on  using Socket
2017-12-26T19:00:08.914024Z    23 Init DB   nowasp
2017-12-26T19:00:08.914075Z    23 Query SELECT 'test connection'
2017-12-26T19:00:08.914153Z    23 Query SELECT cid FROM blogs_table
2017-12-26T19:00:08.914319Z    23 Quit
2017-12-26T19:00:08.914604Z    24 Connect   root@localhost on  using Socket
2017-12-26T19:00:08.914736Z    24 Init DB   nowasp
```

```
2017-12-26T19:00:08.914967Z    25 Connect   root@localhost on
    using Socket
2017-12-26T19:00:08.915024Z    25 Init DB   nowasp
2017-12-26T19:00:08.915186Z    26 Connect   root@localhost on
    using Socket
2017-12-26T19:00:08.915232Z    26 Init DB   nowasp
2017-12-26T19:00:08.915367Z    27 Connect   root@localhost on
    using Socket
2017-12-26T19:00:08.915412Z    27 Init DB   nowasp
2017-12-26T19:00:08.916416Z    26 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                    level_1_help_include_files.
    level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                level_1_help_include_files.
    level_1_help_include_file_key
            WHERE page_help.page_name = 'register.php' ORDER BY
    page_help.order_preference
2017-12-26T19:00:08.916777Z    24 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'Attempting to add account for:
    satan',  now() )
2017-12-26T19:00:08.917615Z    26 Query INSERT INTO accounts (
    username, password, mysignature) VALUES ('satan', '123456',
    'Satan Test Account')
2017-12-26T19:00:08.919651Z    24 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'Added account for: satan',  now()
    )
2017-12-26T19:00:08.920730Z    24 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: register.php',  now
    () )
2017-12-26T19:00:08.921062Z    25 Quit
2017-12-26T19:00:08.921119Z    27 Quit
2017-12-26T19:00:08.921146Z    24 Quit
2017-12-26T19:00:08.921169Z    26 Quit
2017-12-26T19:00:11.825664Z    28 Connect   root@localhost on
    using Socket
2017-12-26T19:00:11.825733Z    28 Init DB   nowasp
2017-12-26T19:00:11.825776Z    28 Query SELECT 'test connection'
2017-12-26T19:00:11.825836Z    28 Query SELECT cid FROM
    blogs_table
2017-12-26T19:00:11.826013Z    28 Quit
2017-12-26T19:00:11.826168Z    29 Connect   root@localhost on
    using Socket
2017-12-26T19:00:11.826214Z    29 Init DB   nowasp
2017-12-26T19:00:11.826355Z    30 Connect   root@localhost on
    using Socket
2017-12-26T19:00:11.826398Z    30 Init DB   nowasp
2017-12-26T19:00:11.826705Z    31 Connect   root@localhost on
    using Socket
2017-12-26T19:00:11.826804Z    31 Init DB   nowasp
2017-12-26T19:00:11.826997Z    32 Connect   root@localhost on
    using Socket
2017-12-26T19:00:11.827048Z    32 Init DB   nowasp
2017-12-26T19:00:11.827686Z    31 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                    level_1_help_include_files.
    level_1_help_include_file_description
```

```
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                level_1_help_include_files.
    level_1_help_include_file_key
            WHERE page_help.page_name = 'login.php' ORDER BY
    page_help.order_preference
2017-12-26T19:00:11.828256Z    29 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: login.php',  now() )
2017-12-26T19:00:11.828975Z    30 Quit
2017-12-26T19:00:11.829030Z    31 Quit
2017-12-26T19:00:11.829586Z    29 Quit
2017-12-26T19:00:11.829625Z    32 Quit
2017-12-26T19:00:20.009520Z    33 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.009641Z    33 Init DB   nowasp
2017-12-26T19:00:20.009738Z    33 Query SELECT 'test connection'
2017-12-26T19:00:20.009856Z    33 Query SELECT cid FROM
    blogs_table
2017-12-26T19:00:20.010056Z    33 Quit
2017-12-26T19:00:20.010261Z    34 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.010362Z    34 Init DB   nowasp
2017-12-26T19:00:20.010538Z    35 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.010579Z    35 Init DB   nowasp
2017-12-26T19:00:20.010682Z    36 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.010714Z    36 Init DB   nowasp
2017-12-26T19:00:20.010964Z    37 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.011067Z    37 Init DB   nowasp
2017-12-26T19:00:20.017749Z    34 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User satan attempting to
    authenticate',  now() )
2017-12-26T19:00:20.018906Z    36 Query SELECT username FROM
    accounts WHERE username='satan'
2017-12-26T19:00:20.019180Z    36 Query SELECT username FROM
    accounts WHERE username='satan' AND password='123456'
2017-12-26T19:00:20.019351Z    36 Query SELECT * FROM accounts
            WHERE username='satan' AND password='123456'
2017-12-26T19:00:20.019575Z    34 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'Login Succeeded: Logged in user:
    satan (27)',  now() )
2017-12-26T19:00:20.020492Z    36 Quit
2017-12-26T19:00:20.020553Z    35 Quit
2017-12-26T19:00:20.020577Z    37 Quit
2017-12-26T19:00:20.020598Z    34 Quit
2017-12-26T19:00:20.027025Z    38 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.027093Z    38 Init DB   nowasp
2017-12-26T19:00:20.027137Z    38 Query SELECT 'test connection'
2017-12-26T19:00:20.027209Z    38 Query SELECT cid FROM
    blogs_table
2017-12-26T19:00:20.027437Z    38 Quit
2017-12-26T19:00:20.027717Z    39 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.027782Z    39 Init DB   nowasp
```

```
2017-12-26T19:00:20.027957Z    40 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.028011Z    40 Init DB   nowasp
2017-12-26T19:00:20.028169Z    41 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.028212Z    41 Init DB   nowasp
2017-12-26T19:00:20.028420Z    42 Connect   root@localhost on
    using Socket
2017-12-26T19:00:20.028514Z    42 Init DB   nowasp
2017-12-26T19:00:20.028613Z    41 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:00:20.029557Z    39 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: /var/www/html/
    mutillidae/home.php',  now() )
2017-12-26T19:00:20.030496Z    40 Quit
2017-12-26T19:00:20.031139Z    39 Quit
2017-12-26T19:00:20.031191Z    42 Quit
2017-12-26T19:00:20.031216Z    41 Quit
2017-12-26T19:00:28.684125Z    43 Connect   root@localhost on
    using Socket
2017-12-26T19:00:28.684210Z    43 Init DB   nowasp
2017-12-26T19:00:28.684258Z    43 Query SELECT 'test connection'
2017-12-26T19:00:28.684333Z    43 Query SELECT cid FROM
    blogs_table
2017-12-26T19:00:28.684510Z    43 Quit
2017-12-26T19:00:28.684803Z    44 Connect   root@localhost on
    using Socket
2017-12-26T19:00:28.684896Z    44 Init DB   nowasp
2017-12-26T19:00:28.685244Z    45 Connect   root@localhost on
    using Socket
2017-12-26T19:00:28.685338Z    45 Init DB   nowasp
2017-12-26T19:00:28.685690Z    46 Connect   root@localhost on
    using Socket
2017-12-26T19:00:28.685778Z    46 Init DB   nowasp
2017-12-26T19:00:28.686073Z    47 Connect   root@localhost on
    using Socket
2017-12-26T19:00:28.686148Z    47 Init DB   nowasp
2017-12-26T19:00:28.686253Z    46 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:00:28.688267Z    46 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                  level_1_help_include_files.
    level_1_help_include_file_description
              FROM page_help
              INNER JOIN level_1_help_include_files
              ON  page_help.help_text_key =
                  level_1_help_include_files.
    level_1_help_include_file_key
              WHERE page_help.page_name = 'captured-data.php'
    ORDER BY page_help.order_preference
2017-12-26T19:00:28.688730Z    46 Query SELECT ip_address,
    hostname, port, user_agent_string, referrer, data,
    capture_date
              FROM captured_data
              ORDER BY capture_date DESC
2017-12-26T19:00:28.690708Z    44 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: captured-data.php',
     now() )
2017-12-26T19:00:28.691652Z    45 Quit
2017-12-26T19:00:28.691728Z    47 Quit
2017-12-26T19:00:28.691756Z    44 Quit
```

```
2017-12-26T19:00:28.691786Z    46 Quit
2017-12-26T19:00:30.874212Z    48 Connect   root@localhost on  using Socket
2017-12-26T19:00:30.874273Z    48 Init DB   nowasp
2017-12-26T19:00:30.874313Z    48 Query SELECT 'test connection'
2017-12-26T19:00:30.874376Z    48 Query SELECT cid FROM blogs_table
2017-12-26T19:00:30.874564Z    48 Quit
2017-12-26T19:00:30.874655Z    49 Connect   root@localhost on  using Socket
2017-12-26T19:00:30.874695Z    49 Init DB   nowasp
2017-12-26T19:00:30.874924Z    50 Connect   root@localhost on  using Socket
2017-12-26T19:00:30.875018Z    50 Init DB   nowasp
2017-12-26T19:00:30.875323Z    51 Connect   root@localhost on  using Socket
2017-12-26T19:00:30.875417Z    51 Init DB   nowasp
2017-12-26T19:00:30.875598Z    52 Connect   root@localhost on  using Socket
2017-12-26T19:00:30.875655Z    52 Init DB   nowasp
2017-12-26T19:00:30.875748Z    51 Query SELECT * FROM accounts WHERE cid='27'
2017-12-26T19:00:30.877088Z    51 Query SELECT * FROM `hitlog` ORDER BY date DESC
2017-12-26T19:00:30.877911Z    51 Query SELECT level_1_help_include_files.level_1_help_include_file_key,
                    level_1_help_include_files.level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                level_1_help_include_files.level_1_help_include_file_key
            WHERE page_help.page_name = 'show-log.php' ORDER BY page_help.order_preference
2017-12-26T19:00:30.879437Z    49 Query INSERT INTO hitlog(hostname, ip, browser, referer, date) VALUES ('::1', '::1', 'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko/20100101 Firefox/57.0', 'User visited: show-log.php',  now() )
2017-12-26T19:00:30.880177Z    50 Quit
2017-12-26T19:00:30.880216Z    52 Quit
2017-12-26T19:00:30.880313Z    49 Quit
2017-12-26T19:00:30.880352Z    51 Quit
2017-12-26T19:00:35.126199Z    53 Connect   root@localhost on  using Socket
2017-12-26T19:00:35.126284Z    53 Init DB   nowasp
2017-12-26T19:00:35.126331Z    53 Query SELECT 'test connection'
2017-12-26T19:00:35.126414Z    53 Query SELECT cid FROM blogs_table
2017-12-26T19:00:35.126625Z    53 Quit
2017-12-26T19:00:35.126773Z    54 Connect   root@localhost on  using Socket
2017-12-26T19:00:35.126835Z    54 Init DB   nowasp
2017-12-26T19:00:35.126976Z    55 Connect   root@localhost on  using Socket
2017-12-26T19:00:35.127033Z    55 Init DB   nowasp
2017-12-26T19:00:35.127170Z    56 Connect   root@localhost on  using Socket
2017-12-26T19:00:35.127209Z    56 Init DB   nowasp
2017-12-26T19:00:35.127347Z    57 Connect   root@localhost on  using Socket
2017-12-26T19:00:35.127385Z    57 Init DB   nowasp
2017-12-26T19:00:35.127445Z    56 Query SELECT * FROM accounts WHERE cid='27'
```

```
2017-12-26T19:00:35.128285Z    54 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: home.php',  now() )
2017-12-26T19:00:35.129310Z    54 Quit
2017-12-26T19:00:35.130070Z    55 Quit
2017-12-26T19:00:35.130117Z    57 Quit
2017-12-26T19:00:35.130146Z    56 Quit
2017-12-26T19:01:01.463371Z    58 Connect   root@localhost on
    using Socket
2017-12-26T19:01:01.463475Z    58 Init DB   nowasp
2017-12-26T19:01:01.463527Z    58 Query SELECT 'test connection'
2017-12-26T19:01:01.463605Z    58 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:01.463803Z    58 Quit
2017-12-26T19:01:01.464003Z    59 Connect   root@localhost on
    using Socket
2017-12-26T19:01:01.464061Z    59 Init DB   nowasp
2017-12-26T19:01:01.464670Z    60 Connect   root@localhost on
    using Socket
2017-12-26T19:01:01.464731Z    60 Init DB   nowasp
2017-12-26T19:01:01.464904Z    61 Connect   root@localhost on
    using Socket
2017-12-26T19:01:01.464995Z    61 Init DB   nowasp
2017-12-26T19:01:01.465150Z    62 Connect   root@localhost on
    using Socket
2017-12-26T19:01:01.465206Z    62 Init DB   nowasp
2017-12-26T19:01:01.465289Z    61 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:01:01.473508Z    61 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                      level_1_help_include_files.
    level_1_help_include_file_description
             FROM page_help
             INNER JOIN level_1_help_include_files
             ON  page_help.help_text_key =
                   level_1_help_include_files.
    level_1_help_include_file_key
             WHERE page_help.page_name = 'add-to-your-blog.php'
    ORDER BY page_help.order_preference
2017-12-26T19:01:01.474348Z    61 Query SELECT * FROM
    blogs_table
             WHERE blogger_name like 'satan%'
             ORDER BY date DESC
             LIMIT 0 , 100
2017-12-26T19:01:01.474595Z    59 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'Selected blog entries for satan',
     now() )
2017-12-26T19:01:01.475676Z    59 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: add-to-your-blog.php
    ',  now() )
2017-12-26T19:01:01.476233Z    60 Quit
2017-12-26T19:01:01.476298Z    59 Quit
2017-12-26T19:01:01.476331Z    62 Quit
2017-12-26T19:01:01.476962Z    61 Quit
2017-12-26T19:01:05.623728Z    63 Connect   root@localhost on
    using Socket
2017-12-26T19:01:05.623802Z    63 Init DB   nowasp
2017-12-26T19:01:05.623853Z    63 Query SELECT 'test connection'
2017-12-26T19:01:05.623958Z    63 Query SELECT cid FROM
    blogs_table
```

```
2017-12-26T19:01:05.624239Z    63 Quit
2017-12-26T19:01:05.624318Z    64 Connect   root@localhost on
    using Socket
2017-12-26T19:01:05.624442Z    64 Init DB   nowasp
2017-12-26T19:01:05.624777Z    65 Connect   root@localhost on
    using Socket
2017-12-26T19:01:05.624902Z    65 Init DB   nowasp
2017-12-26T19:01:05.625236Z    66 Connect   root@localhost on
    using Socket
2017-12-26T19:01:05.625340Z    66 Init DB   nowasp
2017-12-26T19:01:05.625636Z    67 Connect   root@localhost on
    using Socket
2017-12-26T19:01:05.625753Z    67 Init DB   nowasp
2017-12-26T19:01:05.625875Z    66 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:01:05.634138Z    66 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                      level_1_help_include_files.
    level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                level_1_help_include_files.
    level_1_help_include_file_key
            WHERE page_help.page_name = 'view-someones-blog.php'
     ORDER BY page_help.order_preference
2017-12-26T19:01:05.634773Z    66 Query SELECT username FROM
    accounts
2017-12-26T19:01:05.635526Z    64 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: view-someones-blog.
    php',  now() )
2017-12-26T19:01:05.636690Z    65 Quit
2017-12-26T19:01:05.636764Z    66 Quit
2017-12-26T19:01:05.637395Z    67 Quit
2017-12-26T19:01:05.637444Z    64 Quit
2017-12-26T19:01:12.046302Z    68 Connect   root@localhost on
    using Socket
2017-12-26T19:01:12.046377Z    68 Init DB   nowasp
2017-12-26T19:01:12.046526Z    68 Query SELECT 'test connection'
2017-12-26T19:01:12.046668Z    68 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:12.046895Z    68 Quit
2017-12-26T19:01:12.047122Z    69 Connect   root@localhost on
    using Socket
2017-12-26T19:01:12.047188Z    69 Init DB   nowasp
2017-12-26T19:01:12.047520Z    70 Connect   root@localhost on
    using Socket
2017-12-26T19:01:12.047575Z    70 Init DB   nowasp
2017-12-26T19:01:12.047921Z    71 Connect   root@localhost on
    using Socket
2017-12-26T19:01:12.048043Z    71 Init DB   nowasp
2017-12-26T19:01:12.048279Z    72 Connect   root@localhost on
    using Socket
2017-12-26T19:01:12.048339Z    72 Init DB   nowasp
2017-12-26T19:01:12.048416Z    71 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:01:12.049370Z    71 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                      level_1_help_include_files.
    level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
```

```
                level_1_help_include_files.
    level_1_help_include_file_key
            WHERE page_help.page_name = 'view-someones-blog.php'
     ORDER BY page_help.order_preference
2017-12-26T19:01:12.049743Z    71 Query SELECT username FROM
    accounts
2017-12-26T19:01:12.050219Z    71 Query SELECT * FROM
    blogs_table
            WHERE blogger_name like 'stusteiner%'
            ORDER BY date DESC
            LIMIT 0 , 100
2017-12-26T19:01:12.050468Z    69 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: view-someones-blog.
    php',  now() )
2017-12-26T19:01:12.051585Z    71 Quit
2017-12-26T19:01:12.051651Z    70 Quit
2017-12-26T19:01:12.051699Z    72 Quit
2017-12-26T19:01:12.051726Z    69 Quit
2017-12-26T19:01:14.481944Z    73 Connect   root@localhost on
    using Socket
2017-12-26T19:01:14.482087Z    73 Init DB   nowasp
2017-12-26T19:01:14.482143Z    73 Query SELECT 'test connection'
2017-12-26T19:01:14.482220Z    73 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:14.482504Z    73 Quit
2017-12-26T19:01:14.482637Z    74 Connect   root@localhost on
    using Socket
2017-12-26T19:01:14.482700Z    74 Init DB   nowasp
2017-12-26T19:01:14.482995Z    75 Connect   root@localhost on
    using Socket
2017-12-26T19:01:14.483091Z    75 Init DB   nowasp
2017-12-26T19:01:14.483360Z    76 Connect   root@localhost on
    using Socket
2017-12-26T19:01:14.483418Z    76 Init DB   nowasp
2017-12-26T19:01:14.483568Z    77 Connect   root@localhost on
    using Socket
2017-12-26T19:01:14.483611Z    77 Init DB   nowasp
2017-12-26T19:01:14.483695Z    76 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:01:14.484559Z    76 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                    level_1_help_include_files.
    level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                level_1_help_include_files.
    level_1_help_include_file_key
            WHERE page_help.page_name = 'add-to-your-blog.php'
    ORDER BY page_help.order_preference
2017-12-26T19:01:14.485190Z    76 Query SELECT * FROM
    blogs_table
            WHERE blogger_name like 'satan%'
            ORDER BY date DESC
            LIMIT 0 , 100
2017-12-26T19:01:14.485413Z    74 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'Selected blog entries for satan',
     now() )
2017-12-26T19:01:14.486377Z    74 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
```

```
    /20100101 Firefox/57.0', 'User visited: add-to-your-blog.php
    ',  now() )
2017-12-26T19:01:14.486895Z    77 Quit
2017-12-26T19:01:14.486959Z    76 Quit
2017-12-26T19:01:14.487579Z    74 Quit
2017-12-26T19:01:14.487656Z    75 Quit
2017-12-26T19:01:30.014890Z    78 Connect   root@localhost on
    using Socket
2017-12-26T19:01:30.014963Z    78 Init DB   nowasp
2017-12-26T19:01:30.015011Z    78 Query SELECT 'test connection'
2017-12-26T19:01:30.015086Z    78 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:30.015310Z    78 Quit
2017-12-26T19:01:30.015427Z    79 Connect   root@localhost on
    using Socket
2017-12-26T19:01:30.015486Z    79 Init DB   nowasp
2017-12-26T19:01:30.015809Z    80 Connect   root@localhost on
    using Socket
2017-12-26T19:01:30.015892Z    80 Init DB   nowasp
2017-12-26T19:01:30.016097Z    81 Connect   root@localhost on
    using Socket
2017-12-26T19:01:30.016152Z    81 Init DB   nowasp
2017-12-26T19:01:30.016297Z    82 Connect   root@localhost on
    using Socket
2017-12-26T19:01:30.016340Z    82 Init DB   nowasp
2017-12-26T19:01:30.016423Z    81 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:01:30.017469Z    81 Query INSERT INTO blogs_table(
    blogger_name, comment, date) VALUES ('satan', 'now is the
    time for all good men to come to the aid of their country',
     now() )
2017-12-26T19:01:30.018556Z    79 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'Blog entry added by: satan',  now
    () )
2017-12-26T19:01:30.019088Z    81 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                    level_1_help_include_files.
    level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                 level_1_help_include_files.
    level_1_help_include_file_key
            WHERE page_help.page_name = 'add-to-your-blog.php'
    ORDER BY page_help.order_preference
2017-12-26T19:01:30.019843Z    81 Query SELECT * FROM
    blogs_table
            WHERE blogger_name like 'satan%'
            ORDER BY date DESC
            LIMIT 0 , 100
2017-12-26T19:01:30.020062Z    79 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'Selected blog entries for satan',
     now() )
2017-12-26T19:01:30.021008Z    79 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: add-to-your-blog.php
    ',  now() )
2017-12-26T19:01:30.021379Z    80 Quit
2017-12-26T19:01:30.021433Z    82 Quit
2017-12-26T19:01:30.021519Z    79 Quit
```

```
2017-12-26T19:01:30.021554Z    81 Quit
2017-12-26T19:01:31.786764Z    83 Connect   root@localhost on
    using Socket
2017-12-26T19:01:31.786840Z    83 Init DB   nowasp
2017-12-26T19:01:31.786891Z    83 Query SELECT 'test connection'
2017-12-26T19:01:31.786970Z    83 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:31.787202Z    83 Quit
2017-12-26T19:01:31.787333Z    84 Connect   root@localhost on
    using Socket
2017-12-26T19:01:31.787383Z    84 Init DB   nowasp
2017-12-26T19:01:31.787521Z    85 Connect   root@localhost on
    using Socket
2017-12-26T19:01:31.787565Z    85 Init DB   nowasp
2017-12-26T19:01:31.787740Z    86 Connect   root@localhost on
    using Socket
2017-12-26T19:01:31.787794Z    86 Init DB   nowasp
2017-12-26T19:01:31.787921Z    87 Connect   root@localhost on
    using Socket
2017-12-26T19:01:31.787966Z    87 Init DB   nowasp
2017-12-26T19:01:31.788028Z    86 Query SELECT * FROM accounts
    WHERE cid='27'
2017-12-26T19:01:31.788943Z    86 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                     level_1_help_include_files.
    level_1_help_include_file_description
           FROM page_help
           INNER JOIN level_1_help_include_files
           ON  page_help.help_text_key =
                level_1_help_include_files.
    level_1_help_include_file_key
           WHERE page_help.page_name = 'view-someones-blog.php'
     ORDER BY page_help.order_preference
2017-12-26T19:01:31.789327Z    86 Query SELECT username FROM
    accounts
2017-12-26T19:01:31.789806Z    84 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: view-someones-blog.
    php',  now() )
2017-12-26T19:01:31.790806Z    87 Quit
2017-12-26T19:01:31.790817Z    85 Quit
2017-12-26T19:01:31.790873Z    86 Quit
2017-12-26T19:01:31.790973Z    84 Quit
2017-12-26T19:01:36.820845Z    88 Connect   root@localhost on
    using Socket
2017-12-26T19:01:36.820930Z    88 Init DB   nowasp
2017-12-26T19:01:36.820992Z    88 Query SELECT 'test connection'
2017-12-26T19:01:36.821072Z    88 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:36.821201Z    88 Quit
2017-12-26T19:01:36.821402Z    89 Connect   root@localhost on
    using Socket
2017-12-26T19:01:36.821459Z    89 Init DB   nowasp
2017-12-26T19:01:36.821602Z    90 Connect   root@localhost on
    using Socket
2017-12-26T19:01:36.821644Z    90 Init DB   nowasp
2017-12-26T19:01:36.821778Z    91 Connect   root@localhost on
    using Socket
2017-12-26T19:01:36.821820Z    91 Init DB   nowasp
2017-12-26T19:01:36.821945Z    92 Connect   root@localhost on
    using Socket
2017-12-26T19:01:36.821986Z    92 Init DB   nowasp
2017-12-26T19:01:36.822049Z    91 Query SELECT * FROM accounts
    WHERE cid='27'
```

```
2017-12-26T19:01:36.823081Z    91 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                     level_1_help_include_files.
    level_1_help_include_file_description
            FROM page_help
            INNER JOIN level_1_help_include_files
            ON  page_help.help_text_key =
                 level_1_help_include_files.
    level_1_help_include_file_key
            WHERE page_help.page_name = 'view-someones-blog.php'
     ORDER BY page_help.order_preference
2017-12-26T19:01:36.823411Z    91 Query SELECT username FROM
    accounts
2017-12-26T19:01:36.823879Z    91 Query SELECT * FROM
    blogs_table
            WHERE blogger_name like 'satan%'
            ORDER BY date DESC
            LIMIT 0 , 100
2017-12-26T19:01:36.824087Z    89 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: view-someones-blog.
    php',  now() )
2017-12-26T19:01:36.825653Z    90 Quit
2017-12-26T19:01:36.825794Z    92 Quit
2017-12-26T19:01:36.825846Z    91 Quit
2017-12-26T19:01:36.825954Z    89 Quit
2017-12-26T19:01:40.446977Z    93 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.447054Z    93 Init DB   nowasp
2017-12-26T19:01:40.447106Z    93 Query SELECT 'test connection'
2017-12-26T19:01:40.447184Z    93 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:40.447429Z    93 Quit
2017-12-26T19:01:40.447509Z    94 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.447556Z    94 Init DB   nowasp
2017-12-26T19:01:40.447837Z    95 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.447908Z    95 Init DB   nowasp
2017-12-26T19:01:40.448099Z    96 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.448151Z    96 Init DB   nowasp
2017-12-26T19:01:40.448285Z    97 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.448329Z    97 Init DB   nowasp
2017-12-26T19:01:40.456152Z    95 Quit
2017-12-26T19:01:40.456264Z    97 Quit
2017-12-26T19:01:40.456333Z    96 Quit
2017-12-26T19:01:40.456374Z    94 Quit
2017-12-26T19:01:40.462765Z    98 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.462888Z    98 Init DB   nowasp
2017-12-26T19:01:40.463009Z    98 Query SELECT 'test connection'
2017-12-26T19:01:40.463202Z    98 Query SELECT cid FROM
    blogs_table
2017-12-26T19:01:40.463437Z    98 Quit
2017-12-26T19:01:40.463702Z    99 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.463802Z    99 Init DB   nowasp
2017-12-26T19:01:40.464162Z   100 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.464267Z   100 Init DB   nowasp
2017-12-26T19:01:40.464588Z   101 Connect   root@localhost on
    using Socket
```

```
2017-12-26T19:01:40.464687Z   101 Init DB   nowasp
2017-12-26T19:01:40.464995Z   102 Connect   root@localhost on
    using Socket
2017-12-26T19:01:40.465048Z   102 Init DB   nowasp
2017-12-26T19:01:40.465823Z   101 Query SELECT
    level_1_help_include_files.level_1_help_include_file_key,
                    level_1_help_include_files.
    level_1_help_include_file_description
              FROM page_help
              INNER JOIN level_1_help_include_files
              ON  page_help.help_text_key =
                   level_1_help_include_files.
    level_1_help_include_file_key
              WHERE page_help.page_name = 'login.php' ORDER BY
    page_help.order_preference
2017-12-26T19:01:40.466519Z    99 Query INSERT INTO hitlog(
    hostname, ip, browser, referer, date) VALUES ('::1', '::1',
    'Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:57.0) Gecko
    /20100101 Firefox/57.0', 'User visited: login.php',  now() )
2017-12-26T19:01:40.467454Z   100 Quit
2017-12-26T19:01:40.468167Z   101 Quit
2017-12-26T19:01:40.468216Z   102 Quit
```

Appendix B: Complete Listing of SEED SQL Log

Listing B.1: Mutillidae SQL Log: Complete listing of the non-least privilege non-malicious run of the Mutillidae web application.

```
/usr/sbin/mysqld, Version: 5.7.19-0ubuntu0.16.04.1 ((Ubuntu)).
    started with:
Tcp port: 3306  Unix socket: /var/run/mysqld/mysqld.sock
Time                 Id Command    Argument
2018-06-04T23:38:04.148578Z   149 Query set global general_log =
     'ON'
2018-06-04T23:38:09.717757Z   149 Query set global log_output =
    'FILE'
2018-06-04T23:38:44.913379Z   150 Connect   root@localhost on
    Users using Socket
2018-06-04T23:38:44.913532Z   150 Query SELECT id, name, eid,
    salary, birth, ssn, phoneNumber, address, email,nickname,
    Password
       FROM credential
       WHERE name= 'stu' and Password='36
    da2c7673be09d05daa028d25741b0d186913d5'
2018-06-04T23:38:44.913752Z   150 Query INSERT INTO track(ref)
    VALUES ('page_name = unsafe_home.php')
2018-06-04T23:38:44.914121Z   150 Quit
2018-06-04T23:38:49.140716Z   151 Connect   root@localhost on
    Users using Socket
2018-06-04T23:38:49.140912Z   151 Query SELECT id, name, eid,
    salary, birth, ssn, phoneNumber, address, email,nickname,
    Password
  FROM credential
  WHERE name= 'stu'
2018-06-04T23:38:49.141212Z   151 Query INSERT INTO track(ref)
    VALUES ('page_name = unsafe_edit_frontend.php')
2018-06-04T23:38:49.141572Z   151 Quit
2018-06-04T23:39:17.574444Z   152 Connect   root@localhost on
    Users using Socket
2018-06-04T23:39:17.574591Z   152 Query UPDATE credential SET
    nickname='Stu',email='ssteiner@ewu.edu',address='319F CEB',
    Password='36da2c7673be09d05daa028d25741b0d186913d5',
    PhoneNumber='5093594296' where ID=9
2018-06-04T23:39:17.575571Z   152 Query INSERT INTO track(ref)
    VALUES ('page_name = unsafe_edit_backend.php')
2018-06-04T23:39:17.577138Z   152 Quit
```