Popescu Daniel*

HoneyMod Apache honeypot module

Faculty of Computer Science, A. I. Cuza
University of Iasi,
Conferentiar, Dr. Sabin Corneliu Buraga

June, 2016

Springer

*

DECLARAŢIE PRIVIND ORIGINALITATE ŞI RESPECTAREA DREPTURILOR DE AUTOR

Prin prezenta declar că Lucrarea de licenţă cu titlul "*HoneyMod - Apache honeypot module*" este scrisă de mine şi nu a mai fost prezentată niciodată la o altă facultate sau instituţie de învăţământ superior din ţară sau străinătate. De asemenea, declar că toate sursele utilizate, inclusiv cele preluate de pe Internet, sunt indicate în lucrare, cu respectarea regulilor de evitare a plagiatului:

- toate fragmentele de text reproduse exact, chiar şi în traducere proprie din altă limbă, sunt scrise între ghilimele şi deţin referinţa precisă a sursei;
- reformularea în cuvinte proprii a textelor scrise de către alţi autori deţine referinţa precisă;
- codul sursă, imagini etc. preluate din proiecte *open-source* sau alte surse sunt utilizate cu respectarea drepturilor de autor şi deţin referinţe precise;
- rezumarea ideilor altor autori precizează referinţa precisă la textul original.

Iaşi, *29.05.2012*

Popescu Daniel

_____

(semnătura în original)

DECLARAŢIE DE CONSIMŢĂMÂNT

Prin prezenta declar că sunt de acord ca Lucrarea de licenţă cu titlul "*HoneyMod - Apache honeypot module*", codul sursă al programelor şi celelalte conţinuturi (grafice, multimedia, date de test etc.) care însoţesc această lucrare să fie utilizate în cadrul Facultăţii de Informatică.

De asemenea, sunt de acord ca Facultatea de Informatică de la Universitatea Alexandru Ioan Cuza Iaşi să utilizeze, modifice, reproducă şi să distribuie în scopuri necomerciale programele-calculator, format executabil şi sursă, realizate de mine în cadrul prezentei lucrări de licenţă.

Iaşi, *29.05.2012*

Popescu Daniel

_____

(semnătura în original)

Preface

HoneyMod is an Apache module meant to act as a honeypot over any web application running on the Apache structure. It uses certain Apache modules functions to filters both the input to the application and the output of it in such a way that while the original application stays safe, the hacker attacks and modifies a decoy, allowing the honeypot to lure cyberattackers, detect their attacks and study their attempts. This will benefit the application in protecting and exposing it's security breaches.

Iasi Romania, *Daniel Popescu*
June 2016

Contents

1 Introduction 1
- 1.1 Motivation and Goal 1
- 1.2 State of the Art 2
- 1.3 Structure 2
- 1.4 Source Code 3
- 1.5 Acknowledgments 3

2 The Basics 4
- 2.1 Web application security 4
- 2.2 Honeypot 6
- 2.3 Apache 7
- 2.4 Lex&Yacc 9
- 2.5 Web application attacks 10

3 Architecture and Implementation 13
- 3.1 Goals 13
- 3.2 General architecture and Core modules 15
- 3.3 Apache module 16
- 3.4 Reports module 20
- 3.5 Processing the request 21
- 3.6 Treated attacks 25

4 Use Cases 32
- 4.1 Anatomy of an attack 32
- 4.2 Sql injection attack 33
- 4.3 User/Password guessing 35
- 4.4 User Cookie Hijacking 37
- 4.5 XSS Attack 38
- 4.6 Buffer overflow 39
- 4.7 Canonicalization 39
- 4.8 Command execution 39
- 4.9 Attacker story-line 40

5 Conclusions 41

References 42

1

Introduction

HoneyMod aims to classify incoming attacks from hackers and provide them with their desired output without actually compromising the application it supports. It is meant as an overlay, that puts itself between the request and the application, and between the application and the response. For any attacker, the module creates dummy copies, with content that only seems real so that if the attack on the application is successful, the hacker will think his attack succeeded and he may continue his attack phase.
The identification of attackers is currently based on IP address, an effort is made to construct a pattern based on each IP address's attacks so that groups of IP addresses can be assigned to an attacker.

While there are a number of already established honeypots, on different levels of complexity, this one remarks itself by being implemented directly in an Apache module, in a low level environment that allows much more freedom of handling the incoming requests and responses and also manipulating the application itself. Being built on Apache's modular approach, in combination with the ability to hook its filters at the beginning and end of the system's processes, makes this Apache module honeypot easier to configure on any application, and while it doesn't abolish dependencies, it does offer more freedom in that area by default.

As a server-side scripting language for example the module doesn't care if Php/Ruby/Node-js or anything else is used. Also, while at the moment it runs only on the UNIX environment, 90% of the module could already work on windows or any AMP configuration.

1.1 Motivation and Goal

The main motivation for creating this honeypot is the lack of more general and open source honeypots available for the Apache environment.

The goal is to have a functional honeypot that not only logs attacks, but defends the application, exposes security holes to the owner, blacklists attackers and finds patterns in attacks, classifying them so that new types of attacks can be found and also attackers can identified easier.

When completely finished, the application should be easy and fast to configure, and install-able as an Apache module.

1.2 State of the Art

There are quite a few good and complex honeypots that already exist. [12]Glastopf, [13]Specter, [14]KfSensor, [15]HoneyPoint Security Server, [16]Honeyd to name a few. Most offer a big range of attacks that they can handle, but almost all are hard to configure (requiring an expert to install), and limit what you can do with your application and most are also not free.

In the open source field exist a few big organizations that try to gather information and offer help to smaller honeypots, notable between these organizations are the [8]HoneyPot Project and [7]HoneyNet (presented in Chapter 2.3). These offer developers access to their blacklists and attack data and also collect data from thousands of installed honeypots (Honeypot Project has its own install-able honeypot that you can add in your application so that they can collect data).

HoneyMod isn't made to compete at that level, but to be a more easy-to-use application, accessible to a bigger area of developers and focused on the Apache module structure and the way each phase of a request can be analyzed and treated. At the same time it tries to classify attacks and find attack patterns that can hopefully be of help to everyone.

1.3 Structure

We present here the structure of the paper and describe each of the following chapters and important key-notes from each one.

Second chapter will discuss web security principles, explain what a honeypot is and its uses, the HoneyNet and Project Honey Pot projects, present the Apache architecture and key concepts and structures used from it, the Lex&Yacc interpreter and will shortly explain each of the treated attacks.

The third chapter is the core of the paper, it explains the architecture of the application, the Apache module with its more important functionality, as well as the additional example site and admin panel used for reporting. Also here is presented how the application identifies, classifies and treats each of the different attacks.

The fourth chapter presents examples of actual attacks for each treated one, and shows how they are perceived and logged by the honeypot. At the end the path of the attacks is grouped and visualized in the admin panel.

Final chapters are for conclusions based on the reports of the honeypot, listing future endeavors, the bibliography and appendix of the paper.

1.4 Source Code

The source code of the entire project is in 3 separate repositories on GitHub (https://github.com/pdan93) and each can be freely accessed.

- The Apache module https://github.com/pdan93/licenta_apache_module
- The Admin module https://github.com/pdan93/licenta_honeyadmin
- The dummy test site https://github.com/pdan93/licenta_web

The Apache module repository contains detailed instructions on installing it and using the other modules as well.

The code is free software, and can be redistributed under the terms of the GNU Lesser General Public License, version 2.2 as published by the Free Software Foundation. This application is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

1.5 Acknowledgments

I would like to thank my coordinator, Lect. Dr. Sabin-Corneliu Buraga, for his support in this project, always quick in offering help and pointing me in the right direction.

Though not aware, many other people contributed along the way in the making of this project. Special thanks to Kim Guldberg, who also pointed me in the right direction and made sure of the attackers I aim at. Special appreciation goes to the other professors of the Faculty of Computer Science, A. I. Cuza University of Iasi.

Special thanks to the author of "The Apache Modules Book"[2] Nick Kew, for writing an incredible introduction and companion in developing Apache modules, especially when no other documentation comes even close to the in depth explanations and logical presentations of concepts and structures vital in this application.

Lastly I express my appreciation to the amazing open source community, without which I would still be struggling in the development phase. Thanks to the people behind so many excellent and useful open source tools: Apache, Honey Pot Project, [17]OWASP, [18]OWASP's WebScarab, [19]Burp Suite, [20]Generate Data com and many others.

2

The Basics

2.1 Web application security

Web application security is a branch of Information Security that deals specifically with security of websites, web applications and web services. At a high level, web application security draws on the principles of application security but applies them specifically to Internet and Web systems.

Web application suffer from security attacks more then any other type of application. Main reason for that is that websites and the applications that exist on them are in a sense the virtual front door for all corporations and organizations.

Web application security is a subset of Software security. A common problem and the cause for many security holes in applications is treating security as a feature, or a collection of technologies. Security is a property of a system, it is created based on all aspects of the application. Paco Hope & Ben Walther say in [1] "No amount of authentication technology, magic crypto fairy dust, or service-oriented architecture will automatically solve the security problem". They go on to say "In fact, security has more to do with testing and assurance than anything else".

2.1.1 Why attack web applications

The motivations for hacking are numerous and have been discussed at length for many years in a variety of forums. We will point out some of the features of web applications that make them so attractive to attackers. Understanding these factors leads to a much clearer perspective on what defenses need to be put in place to mitigate risk.

- Ubiquity - Web applications are almost everywhere today and continue to spread rapidly across public and private networks. Web hackers are unlikely to encounter a shortage of juicy targets anytime soon.
- Simple techniques - Web app attack techniques are fairly easy understood, even by the layperson, since they are mostly text-based. This makes manipulating application input fairly trivial.
- Anonymity - The internet still has many unaccountable regions today, and it is fairly easy to launch attacks with little fear of being tracked.

- Bypasses firewalls - Inbound HTTP/S is permitted by most typical firewall policies.
- Custom code - Because of the popularity of web development platforms like ASP.NET and LAMP, most web applications are assembled by developers who have little experience.
- Money - with web applications having risen to immense fortunes, the motivation for web hacking has itself moved from fame to fortune.

2.1.2 Web application security testing

Web security testing is using a variety of tools, both manual and automatic, to simulate and stimulate the activities of our web application. Malicious inputs like cross-site scripting attacks are submitted using both manual and scripted methods to the web application. Malicious SQL inputs are used in the same way. Among the boundary values of security testing things like predictable randomness and sequentially assigned identifiers are considered to make sure that common attacks using those values are thwarted.

It is the goal of web application security testing to produce repeatable, consistent tests that fit into the overall testing scheme, but that address the security side of web applications.

2.1.3 Web application hacking

In [5] it is said that "the essence of web hacking is tampering with applications via HTTP" and that there are "three simple ways to do this":

- Directly manipulating the application via its graphical web interface
- Tampering with the Uniform Resource Identifier, or URI
- Tampering with HTTP elements not contained in the URI

2.1.4 OWASP Top 10

The Open Web Application Security Project (OWASP) is a worldwide not-for-profit charitable organization focused on improving the security of software.

The OWASP Top Ten is a powerful awareness document for web application security. The OWASP Top Ten represents a broad consensus about what the most critical web application security flaws are. Project members include a variety of security experts from around the world who have shared their expertise to produce this list.

Many of the attacks present in the Top 10 are treated in this paper, as such we list here shortly the 2013 Top 10 for a better overall understanding of the current web application attacks and classifications of an attack. Some of these will be explained in detail in this paper.

1. Injection
2. Broken Authentication and Session Management
3. Cross-Site Scripting(XSS)
4. Insecure Direct Object Reference
5. Security Misconfiguration
6. Sensitive Data Exposure
7. Missing Function Level Access Control
8. Cross-Site Request Forgery (CSRF)
9. Using Components with Known Vulnerabilities
10. Unvalidated Redirects and Forwards

2.2 Honeypot

A honeypot is a computer system that is set up to act as a decoy to lure cyberattackers, and to detect, deflect or study attempts to gain unauthorized access to information systems.

This ties in with the last chapter, one way to test for security is to watch how hackers might attack your application.

Viewing and logging this kind of activity can provide an insight into the level and types of threat a network infrastructure faces while distracting attackers away from assets of real value.

Though a honeypot helps in understanding and deflecting incoming attacks, it must not be seen as a replacement for a standard IDS(Intrusion detection system). A honeypot is to be closely managed and observed.

In [6] it is said that in general there are two popular reasons behind setting up a Honey Pot

1. Learn how intruders probe and attempt to gain access to your systems. The general idea is that since a record of the intruders activities is kept, you can gain insight into attack methodologies to better protect your real production systems.
2. Gather forensic information required to aid in the apprehension or prosecution of intruders. This is the sort of information often needed to provide law enforcement officials with the details needed to prosecute.

2.2.1 HoneyNet and Project Honey Pot

HoneyNet and Project Honey Pot are two of the biggest projects that focus on the idea of a network of shared honeypots.

HoneyNet focuses on discovering "the tools, tactics and motives involved in computer and network attacks" and sharing the "lessons learned". Based on [7] their vision is: The Honeynet Project is a diverse, talented, and engaged group of international computer security experts who conduct open, cross disciplinary research and development into the evolving threat landscape. It

cooperates with like-minded people and organizations in that endeavor.

Project Honey Pot is a distributed system for identifying spammers and spambots. It offers access to a blacklist API and offers it's own honeypot software which can be installed on any of your websites to participate in the project. In [8] the community describe their actions as: We collate, process, and share the data generated by your site with you. We also work with law enforcement authorities to track down and prosecute spammers. Harvesting email addresses from websites is illegal under several anti-spam laws, and the data resulting from Project Honey Pot is critical for finding those breaking the law.

2.3 Apache

2.3.1 The Apache HTTP Server

The Apache HTTP Server Project [21] is an effort to develop and maintain an open-source HTTP server for modern operating systems including UNIX and Windows. The Apache HTTP Server ("httpd") was launched in 1995 and it has been the most popular web server on the Internet since April 1996. The Apache HTTP Server is a project of The Apache Software Foundation.

One of the key factors in making Apache unique and powerful is explained in [2] The major innovation in Apache2, which transforms it from a "mere" webserver (like Apache 1.3 and others) into a powerful applications platforms, is the filter chain. Filters enable a far cleaner and more efficient implementation of data processing than was possible in the past, as well as separating content generation from its transformation and aggregation.

It was chosen for use in this application due to it's popularity and modular structure.

2.3.2 The Apache Platform and Architecture

Apache runs as a permanent background task: a daemon (UNIX) or service (Windows). Start-up is a slow and expensive operation, so it usually starts at system boot and remains permanently up.

The Apache HTTP Server comprises a relatively small core, together with a number of modules, in addition the server relies on the Apache Portable Runtime (APR) libraries, which provide a cross-platform operating system layer and utilities, so that modules don't have to rely on non-portable operating system calls.

2.3.3 Basic Concepts and Structures used throughout the application

To understand an Apache module, an overview of the basic units of webserver operations and the core objects that represent them within Apache is needed. Most important here are:

- the server (server_rec)
- the TCP connection (conn_rec)
- and the HTTP request (request_rec)

each defined in the httpd.h header file, and also extensively documented on the Apache2 documentation [9]

Also used as a core concept in Apache are APR pools (apr_pool_t). They are the core of resource management in Apache. Whenever a resource is allocated dynamically, a cleanup for it is registered, ensuring that system resources are freed when no longer used.

ap_ header files generally define low-level API elements.

apr_ header files define the APR APIs. They are external but essential to the webserver. Notable here are Buckets (apr_bucket) and Brigades (apr_bucket_brigade), they form a ring structure with diverse information and are the principal way of passing data through all of Apache's hooks and filters.

2.3.4 Request Processing in Apache

Our honeypot application is build in the Request Processing structure of Apache, therefore it needs to be explained. The architecture of Apache's webserver is very similar to that of a general-purpose one, it accepts a Request, applies a Content Generator and sends the Response.

[2] Most, though by no means all, modules are concerned with some aspect of processing an HTTP request. But there is rarely, if ever, a reason for a module to concern itself with every aspect of HTTP-that is the business of the httpd[10]. The advantage of a modular approach is that a module can easily focus on a particular task but ignore aspects of HTTP that are not relevant to it.

2.3.5 Apache filters and hooks

Because Apache filters are the ones that get to process the input and output data of the Content Generator they are perfect for our honeypot. There are different types of content filters that can be handled therefore they are classified as: Content Filters (AP_FTYPE_RESOURCE and AP_FTYPE_CONTENT_SET), Protocol Filters (AP_FTYPE_PROTOCOL) and Connection Filters (AP_FTYPE_TRANSCODE, AP_FTYPE_CONNECTION and AP_FTYPE_NETWORK).

Different hooks can be set for different parts of the request process, giving us access to anything we need. Most used and common hook is ap_hook_handler which "Create a hook in the request handler, so we get called when a request arrives" but there are other useful hooks as of Apache 2.4:

- ap_hook_child_init: Place a hook that executes when a child process is spawned (commonly used for initializing modules after the server has forked)

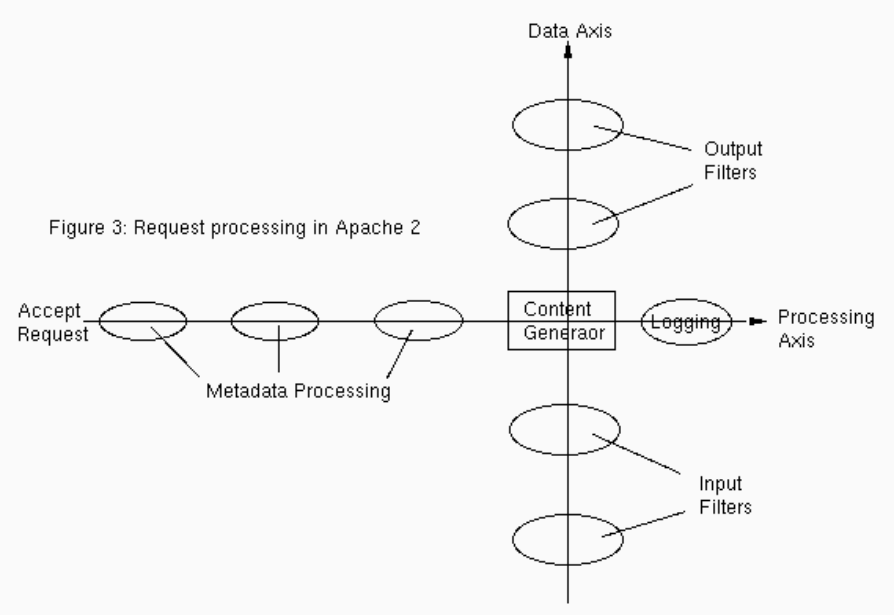

Fig. 2.1. The Data Axis and Filters as explained by Apache Tutor

- ap_hook_pre_config: Place a hook that executes before any configuration data has been read (very early hook)
- ap_hook_post_config: Place a hook that executes after configuration has been parsed, but before the server has forked
- ap_hook_translate_name: Place a hook that executes when a URI needs to be translated into a filename on the server (think mod_rewrite)
- ap_hook_quick_handler: Similar to ap_hook_handler, except it is run before any other request hooks (translation, auth, fixups etc)
- ap_hook_log_transaction: Place a hook that executes when the server is about to add a log entry of the current request

2.4 Lex&Yacc

2.4.1 Lex - A Lexical Analyzer Generator

[3] Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which

match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

2.4.2 Yacc: Yet Another Compiler-Compiler

[4] Computer program input generally has some structure; in fact, every computer program that does input can be thought of as defining an "input language" which it accepts. An input language may be as complex as a programming language, or as simple as a sequence of numbers. Unfortunately, usual input facilities are limited, difficult to use, and often are lax about checking their inputs for validity.

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that han- dles the input process; frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine.

2.5 Web application attacks

The application treats some of the most common web application attacks, as presented in this chapter. Web application attacks are extremely diverse, because of that no one pattern can be established to detect or describe them, but a combination of many.

2.5.1 Profiling

Profiling is the first aspect of web hacking that attackers focus on. Profiling defines the tactics used to research and pinpoint how web sites are structured and how their applications work.

This application tries to identify possible profiling by identifying Brute Force repeated attacks, requests that may not have any apparent malicious intent but are made in a short time span in order to map the entire structure of the website.

2.5.2 SQL injections

SQL injection is a technique where malicious users can inject SQL commands into an SQL statement, via web page input. If the input isn't validated, injected SQL commands can alter SQL statements and compromise the security of a web application.

A complex, extremely varied and very easy to use attack, it is the number one attack in almost every web security list. Because it has many formats, an SQL injection's threat varies from harmless to extremely dangerous.

[11] The risk of SQL injection exploits is on the rise because of automated tools. In the past, the danger was somewhat limited because an exploit had to be carried out manually: an attacker had to actually type their SQL statement into a text box. However, automated SQL injection programs are now available, and as a result, both the likelihood and the potential damage of an exploit has increased enormously.

2.5.3 Broken Authentication and Session Management

Application functions related to authentication and session management are often not implemented correctly, allowing attackers to compromise passwords, keys, or session tokens, or to exploit other implementation flaws to assume other users identities. From this category the application focuses on:

- User/Password guessing - repeated attack on a login page to guess the user and password of an user using brute-force type algorithm.
- Cookie Attack - changing the value of cookies in an attempt to either hijack a session or gain access to unauthorized information.

2.5.4 Cross-Site Scripting (XSS)

XSS flaws occur whenever an application takes un-trusted data and sends it to a web browser without proper validation or escaping. XSS allows attackers to execute scripts in the victims browser which can hijack user sessions, deface web sites, or redirect the user to malicious sites.

It happens anytime a website doesn't filter user input properly, whether in a form field, a URL parameter, poor use of header information, and several other factors. The application focuses on JavaScript-based XSS attacks.

2.5.5 Security Misconfiguration

Good security requires having a secure configuration defined and deployed for the application, frameworks, application server, web server, database server, and platform. Secure settings should be defined, implemented, and maintained, as defaults are often insecure. Additionally, software should be kept up to date.

As this depends on the specific platforms and technologies used, the application only focuses on three more general attack types:

- Buffer Overflow - sending too much information to the application with the hope of escaping/breaking validation.

- Command execution - command execution is a common goal for an attack because command-line access quickly leads to a full compromise of the web server.
- Canonicalization - The basic form of this attack is to move outside of the web document root in order to access system files.

3

Architecture and Implementation

In this chapter we discuss the architecture of the application, the way it was build and how it treats some of the attacks. We start by introducing the main goals of the entire honeypot. Modules will be described in detail and a focus will be put on the classification of a web request and the various ways attacks are treated.

3.1 Goals

In this section we present the goals that should be met in order to make the HoneyMod useful in outside hands. Please note that the goals given here are set for the complete honeypot, and are not all met at the time of this thesis.

3.1.1 Platform independence

Platform independence is extremely desirable in large-scale thus highly-heterogeneous environments, such as the Internet. While currently the honeypot is tested and works in the Ubuntu Server environment, a proper platform independence would mean the ability of the honeypot to work on any UNIX server and also on any AMP stack.

3.1.2 Content generator independence

A Content generator is considered in Apache any software that generates a response content based on a HTTP request. The honeypot supports any content generator, be it PHP, NodeJS, Ruby or any other language. It should also support general attacks on each of the more common languages.

3.1.3 Extensibility

The ability to easily add new functionalities, or reimplement existing ones using different technologies, without impacting the system as a whole, makes a system extensible. Since it's based on Apache's modular philosophy, the application should also embrace it and become a modular system, to allow new attacks and functionalities to easily be added.

3.1.4 Easy to use

The application is meant for beginner and middle level developers, therefore it should be easy to use, with a simple and friendly user interface and an as clear as possible information on the reports and logs of the attacks.

3.1.5 Easy to install

Being an Apache module, the application will be install-able by a simple command in UNIX, and will come in a simple package. The Reports admin module will be available as a simple PHP website that is easy to configure and only requires basic knowledge.

3.1.6 Helpful

The main goal of a honeypot is to detect, deflect and study attack attempts. This honeypot aims to detect if the protected application that it protects has a vulnerability or not for each attack that is made. Therefore the biggest help it will offer a developer, is the ability to test and find security holes.

3.1.7 Security

Being an application focused on security and simulating security holes to attackers, the application itself should be extremely secure, itself and also in it's actions. The main objectives to be achieved by the honeypot should be:

- access control - restricting access to privileged entities and actions.
- confidentiality - keeping information about itself and about the protected site secret from all but those who are authorized to see it.
- data integrity - ensuring the live data is not altered by unauthorized or unknown means.
- non-repudiation - preventing the denial of previous commitments or actions

3.1.8 Pragmatism

While the theoretical foundations of the platform are important enough, more important is the practical applicability of the theoretical methods in the actual implementation. Some good pragmatic rules are:

1. No redundant components
2. Fix bad designs, wrong decisions and poor code as soon as found
3. Crash Early - error codes are good but the application should be interrupted when a big problem occurs
4. Use as many configurable files and no hard-coding.

3.2 General architecture and Core modules

The main application is build on top of the hooks and filters of Apache. The diagram presented in Fig. 3.1. represents the general architecture of the entire HoneyMod application which we will now discuss.

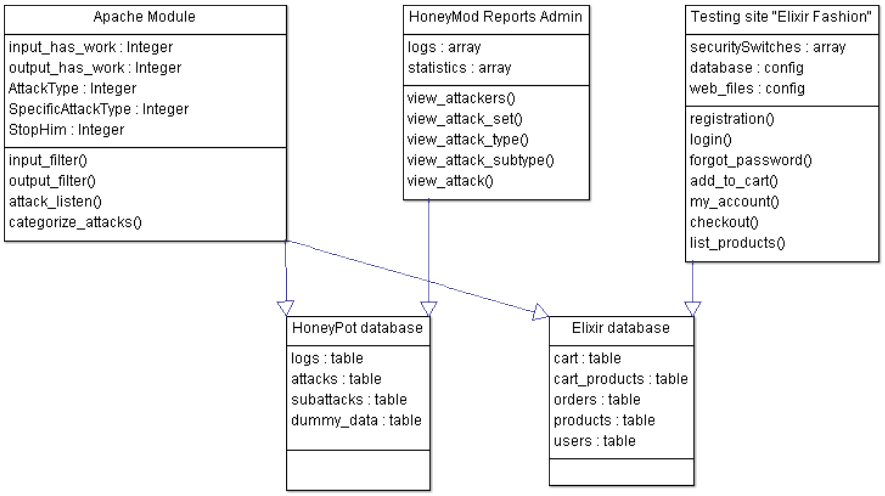

Fig. 3.1. General architecture UML Diagram

While the Apache module is the core of the entire application, each other section has its own uses inside the honeypot.

The Apache module is the one that filters the input of the request and, if necessary, the output of the content generator. It also listens for an attack and tries to classify it. Once classified it treats the attack accordingly, this will be discussed in length in the following chapters about Apache module and Treated attacks. What is essential here, is that the Apache module can read and alter the entire request of the attacker, when necessary also changing the URL path to a copy of the live protected website. This means the attacks are logged without a problem and the attack reaches just a copy, meaning if the attack is successful the attacker sees an appropriate result but the live site isn't damaged.

The Apache module needs its own database for logging the requests and all conclusions taken for each of them, for the classification of the attacks and it also provides a data table with thousands of generated dummy information that is used in the cloned databases instead of the actual data.

To make sense of the logs and the classified data from the honeypot database an admin section has been constructed, with a graphical interface, named HoneyMod Reports Admin. This provides accessible functions so that

an administrator can see the latest attacks, check for new found security holes and see the classified attacks and logged visits.

The HoneyMod honeypot was constructed initially as an research focused application, to classify and see how hackers attack, but because of the way some of the attacks can be treated it can be used as a production honeypot as well.

For the research part a testing website was constructed. It is a small e-commerce website called "Elixir Fashion" focused on selling t-shirts. It has a registration area, an "my account" area and a checkout form for the products to be bought. In all the areas that the site can be attacked the code has certain security switches. For example in the "forgot password" form a switch determines if the input sent by the client is parsed with mysqli_escape_string so that any SQL injection attack is nullified or not. These switches are later used with random or set values of on/off for the copies the hackers attack.

The site comes with its own database, which is also used by the Apache module in its cloning and filled with dummy data.

3.3 Apache module

3.3.1 General architecture

The Apache module is composed of a main c file called mod_honeymod.c and a few header files (module_helpers.h, module_logs.h, module_sql.h, module_attacks.h, module_classifiers.h) each containing functions used at one point throughout the process.

- module_helpers.h - contains small helping functions to aid in redundant tasks as copying strings, searching in strings, bucket writing, APR table handling, reading files.
- module_logs.h - contains functions that mainly aided in the debugging of the program, and also some logging functions of the request data.
- module_sql.h - contains any function that does operations with the SQL database. Cloning the database, logging the request data and other attack specific SQL queries.
- module_attacks.h - contains functions that listen for attacks, main classifier function and other attack specific functions.
- module_classifier.h - contains functions that use Lex and YACC to match request bodies to a pattern.

Overall, the actions taken in the module are: input and output filters are registered, input initial function is called, input filter is called, at the end of the input filter if there is an attack supplementary actions are taken (classifier is called, ip is checked, attack is treated, yacc and neuronal network functions are called and if needed certain output filter switches are turned on so that the output is modified.), output filter is called which reads the entire output

and either leaves it alone or can modify it. This process is described in Fig. 3.2.

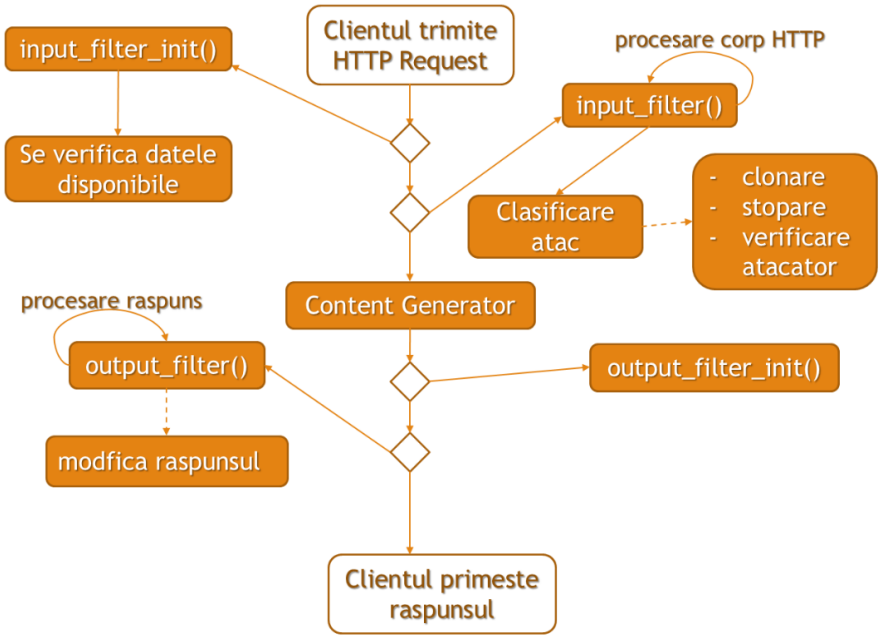

Fig. 3.2. Apache Module Activity Diagram

3.3.2 Filter Functions

After being registered with the ap_register_input_filter (or output) function, a filter will be called for any request. Because Apache uses a Pipeline format, and also data is contained in apr_bucket_brigade structures, to take the whole input content means an elaborate process:

The filter function may be called more times then one depending on the length of the data, and the data itself may be in more then one bucket. Therefore the general structure of a code that handles this is:

```
apr_bucket* b;
char* buf;
size_t bytes;
  for ( b = APR_BRIGADE_FIRST(bb); b != APR_BRIGADE_SENTINEL(bb);
   b = APR_BUCKET_NEXT(b) )
      {
      if ( APR_BUCKET_IS_EOS(b) ) {
```

```
        //end of the line
        }
        else
        {
        apr_bucket_read(b, &buf, &bytes, APR_BLOCK_READ);
        }
      }
```

Of course our filters may have other code before and after this, input filters for example require additional handling, but this general algorithm allows us to gather all the data and process it.

3.3.3 Attack listen and categorization

In the Attack Listen phase we look at the request and decide if it is close to being an attack or is just a normal client visit. If the request method is POST then we automatically treat it as a potential attack. If the url contains GET parameters or more then a normal number of requests was made by the client in a set time then again we may have an attack.

Categorization of the attack can only be done after the input filter gathered the request body data (most of the other info is already in the request_rec structure present everywhere). Initial regex patterns are tested against the body to try and classify the type of attack we are dealing with. If that succeeds then the appropriate attack handler is called (ex: categorize_sql_injection()). If not, then we continue with other verifications such as looking at the past requests to see if the attack may be a buffer-overflow type attack or a brute-force one. In the end 3 global variables can be modified in this phase:

- AttackType - the general attack type (SQL injection, name/password guessing etc...)
- SpecificAttackType - a more specific sub-categorization of the attack. For example SQL injections are defined by: 1 - tautologies, 2 - illegal/logically incorect queries, 3- union query, 4- piggy backed query, 5-stored procedures, 6- alternate encodings
- StopHim - if this is set to 1, it means that the attack was classified as a very dangerous one and the request must not be allowed to reach the content generator or (in case of buffer overflow type attacks) we've had enough of this attacker.

3.3.4 Sql database cloning

At first, a different approach was tested. Starting with SQL injections for example, the output was to be modified based on the attack input. This approach proved too complex and good only for the research test site.

The best way to make the hacker think his attacks are successful is to let

them actually be successful, therefore each attacker gets a clone of the actual database, without the real data, but fake randomly generated dummy data instead. That way attack like SQL injection, name/password guessing or XSS can be treated more generally, allowing us to observe a more natural path of the hacker. Certain limits and precautions are put in place of course:

1. The cloned database doesn't use the real data but dummy generated one.
2. Each database gets its own user with access only to that database, blocking the attacker's possibility of harmful attacks such as SHUTDOWN; in SQL
3. Each database has a size limit in case the attacker finds a way to insert data into it.
4. A cloned database is created only when an attack is of an certain level, so the number of cloned databases is not too big.
5. Each clone has a certain lifespan and isn't allowed to exist and occupy memory forever.

Besides the rest of the code, these 3 commands are the most relevant ones:

```
sprintf (query + strlen(query)
  ,"CREATE DATABASE elixir_fashion%d"
  ,last_sql_clone_nr);
sprintf(query + strlen(query)
  ,"GRANT ALL PRIVILEGES ON elixir_fashion%d.* To
  'test%duser'@'localhost' IDENTIFIED BY 'test%dpass';"
  ,last_sql_clone_nr
  ,last_sql_clone_nr
  ,last_sql_clone_nr);
sprintf(query + strlen(query)
  ,"mysqldump -h localhost -u root -p'xxxxxxx' elixir_fashion |
  mysql -h localhost -u test%duser -p'test%dpass' elixir_fashion%d"
  ,last_sql_clone_nr
  ,last_sql_clone_nr
  ,last_sql_clone_nr);
```

3.3.5 Request logging

Each client request is logged right to the last detail. While the list of logged column items is 60+ we'll present here a few of the more important ones, the others being any possible string or integer values of the request_rec, conn_rec, server_rec, request_rec->useragent_addr, conn_rec->client_addr, server_rec->process structures.

1. id - id of the log
2. is_attack - categorized as attack or not
3. attack_type and specific_attack_type - categorization of the attack
4. timestamp - date and time of the request
5. r_method - request method

6. r_user - if any user was sent in request
7. r_uri - parsed url
8. r_filename - file that the url points to
9. r_useragent_ip - IP address of the client
10. s_port - port on which the request is made
11. headers_in - request headers
12. headers_out - response headers
13. request_body - complete request body if any
14. user_post and pass_post - used for name/password guessing attacks

3.3.6 IP address verification

Each client's IP address is verified in his first visit in two steps, firstly it's matched against a blacklist database, and secondly against a proxy database. The results are logged in the ips table.

To verify if the IP address is on any blacklist we use a small PHP script that makes a dns_get_record call to the HttpBl API. The PHP script is called through a system call from the C code. HttpBl belongs to "Project Honey Pot", one of the largest online tracking communities, that, by its name deals mostly with managing and collecting data from honeypots. The DNS record call returns a threat amount and a classification of the IP address (Suspicious, Harvester, Comment spammer, and combinations).

Secondly, the IP address is also verified to see if the attacker uses a known proxy by calling a proxy verify API. If the result is positive data about the proxy is given in the response.

3.4 Reports module

To clearly see the results of the honeypot, and to be able to make conclusions based on it, an user interface was created that handles the data from the honeypot database.

The module is written in PHP, is called HoneyPot Admin, and is available at root_url/honeyadmin/. To build the user interface, a free open-source admin HTML Theme called "AdminLTE" was used.

The admin shows different statistics on the dashboard, and offers links to more in depth logs for different attacks or requests. The dashboard is presented in Fig. 3.3.

Some useful statistics are displayed in the dashboard for quick viewing: number of attacks, unique visitors, unique attackers, percentage rate of attackers, a graphic with the number of attacks and visits on each day, and a donuts type chart with the percentage and number of the different types of attacks. A table with the latest attacks grouped by IP address and attack type is also displayed for convenience.

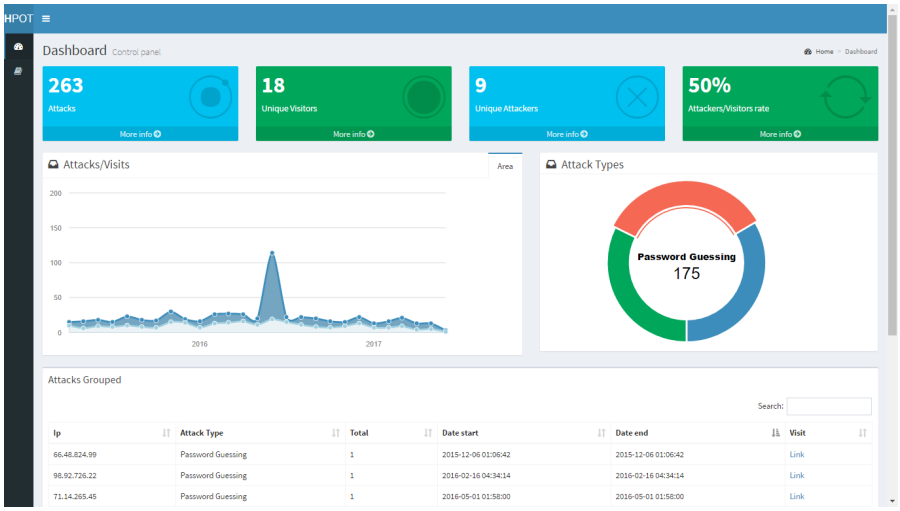

Fig. 3.3. HoneyPot Admin dashboard

Links to different other sections of the admin are available in the left menu and are described below:

- dashboard - links back to the home page of the admin
- attack types - page that displays statistics for each attack type and its specific sub-types.
- attacks - displays a table with the attacks grouped by attack type and IP address, and also may display a visual representation of a certain grouped attack.
- documentation - a small documentation of the admin, and links to this paper and the git repositories of the honeypot.

3.5 Processing the request

To classify an attack a request needs to be analyzed in detail. In some cases a single request isn't enough and we need to look to a group of requests. Each attack has its own pattern of course, but we can essentially assign them to two categories:

1. Single Request - such an attack can be recognized looking only at the current request (XSS, SQL injections, file permissions)
2. Multiple Request - attacks that have a result only after an undefined number of requests (brute force attacks, user/password guessing, session hijacking)

In the same manner some attacks can be detected by looking at the request body, while some don't need access to the request body. This matters because

while we have access to the rest of the request almost at any point in multiple hooks, the request body can only be processed in the input filter. Therefore 2 categories emerge based on this logic as well:

1. Request Body - attacks that come in the body of the request (XSS, SQL injections)
2. Request Info - attacks that don't necessarily use the request body but use headers or GET parameters.

Knowing this, we can now look at the different functions and their place in the module. As discussed in the Apache module chapter, an input filter is registered through the ap_register_input_filter function, which actually allows us to have 2 functions called, an initial one called once, and the actual filter function. This makes it easy for us to separate the "Request Body" classifications from the "Request Info" classifications.

```
ap_register_input_filter("honeymod_input_filter",
input_filter, //the actual filter function
     input_filter_init, //the initial function
     AP_FTYPE_CONTENT_SET);
```

Essentially some initial verification is made in the initial function, and some attacks can be checked in here as well, and then in the filter function, after the complete body has been parsed we do extra verification for other attack types and if necessary change environment variables.

3.5.1 Initial Verification - input_filter_init()

The input_filter_init() is the start of our application, meaning the request makes first contact with us here. The function first sets the default values for the global variables. This is done because Apache runs continuously and our module isn't restarted for every request, meaning in some cases after one request our global variables may be different when we get to the input_filter_init() function.

We then call attack_listen() which we'll summarize below and explain certain parts of it:

```
int attack_listen(ap_filter_t* f) {
...
FILE * ip_map = fopen("/var/www/ip_map","r");
... //(1)

int ok=0; //(2)
if (my_regex("(POST|PUT)",f->r->method))
ok=1; //(3)

if (AttackType==0)
```

```
{
struct MCookie setcookie = get_last_set_cookie(f->r);
... //(4)
}
if (ok==1)
{ //(5)
input_has_work = 1;
output_has_work = 1;
input_buffer = -1;
}
else
{
if (hasOwnDb) //(6)
changefilepath(f->r);
}
...
}
```

1. We search to see if the IP address has already attacked once. If it has, it means that database and file clones exists for it and we register that in the hasOwnDb and OwnDbNr global variables.
2. the ok variable will be 0 if we do not think the current request is an attack
3. if the request method is POST or PUT, it means we have to look into the request body
4. here we look for attacks on cookies.
5. a possible attack means both input and output filters have work to do
6. if there is a file clone for this IP address, we change the request_rec->filename variable so that the attacker visits that file. For example when visiting the index.php page, request_rec->filename has the value "/var/www/html/index.php" and we change that to "/var/www/html/copy10/index.php" where 10 is an identifier for that IP address.

3.5.2 Filter Verification - input_filter()

We have already described the anatomy of a filter in chapter 3.2.2, we will focus now on the actions and decisions the filter does. The filter reads the input data only if the global variable input_has_work is 1.

After reading the entire request body the categorize_attack() function is called. This is where all other attacks are verified and categorized. After that an additional verification is made, in case the attack has been categorized as dangerous, the filter stops the request from going forward to the content handler.

We will now describe the actions taken in the categorize_attack() function:

```
int categorize_attack(request_rec* r) {
...
```

```
if (strlen(input_buffer)>0 && AttackType==0)
{
if (my_regex("...",input_buffer)) //(1)
{
AttackType = 1;
...
}
else
if (my_regex("...",input_buffer)) //(2)
{
AttackType=4;
}
else
{
... //(3)
}
}
if (AttackType>0)
{
verify_ip(r); //(4)
if (SpecificAttackType>0 && hasOwnDb==0)
clonedb(r); //(5)
}
...
}
```

1. we verify for SQL injections, if that is the case, a function that classifies SQL injections is called.
2. we verify for XSS, if that is the case, a function that classifies XSS is called.
3. we verify for user/password guessing.
4. as the request is now an attack, the IP address is verified against the Blacklist and Proxy API's
5. if this is the first attack from this IP address, database and files are cloned.

3.5.3 Lex&Yacc interpreter for SQL injections

As the SQL injection attack is one of the most diverse and complex attack, having many different forms, an effort was made in classifying this special class of attacks, more then on any other attack. While useful, simple regular expressions weren't enough in this regard. Thus the use of a more complex environment that uses regular expressions as tokens in combination with a grammatical approach was the key.

Lex in combination with Yacc were used to parse the SQL injection and

classify it more clearly. While this also doesn't offer a 100% correct classification, it does improve on the simple regex use and there is a possibility to be greatly improved in the future.

Lex was used to recognize and create tokens from certain groups of characters like: STRING, NUMBER, COMPARATOR, LOGICAL and so on.

Yacc then uses these tokens and tries to separate the attacks into different classes. A relevant example for this is the start of the Yacc grammar:

```
injections : injection
| injection injections
;
injection : tautology
| illegal
| union
| piggy_backed
| stored_procedure
| alternate_encoding
;
```

As we can see the SQL injection is matched against 6 non-terminals. These are each defined in detail, with the use of other non-terminals, terminals and tokens.

3.6 Treated attacks

In this chapter we will describe in depth the current list of treated attacks, how they work and how we find and treat them.

3.6.1 SQL injections

In general SQL injection attacks are made progressively, starting from small probing ones to some that can have catastrophic consequences. Such a scenario will be presented in the "Use Cases" chapter.

A general accepted categorization is:

1. tautologies - attacks that aim to make the WHERE clause TRUE no mater what. (x' or 'x'='x)
2. illegal/logically incorrect queries - these aim to break the query, and are usually used for probing. (")
3. union queries - these use the UNION or JOIN functionality to select more fields, possibly from other tables (UNION SELECT * FROM...)
4. piggy backed queries - these interrupt the query and follow with a different query. (; UPDATE ...)

5. stored procedures - a form of the piggy backed query, but dangerous enough to be in its own category, this method means the attacker calls procedures for the database or even possibly the entire MySQL system. (; SHUTDOWN;..)
6. alternate encoding - another dangerous type, this conceals the attack encoding a command in such a way that the parser sees it as text, but MySQL decodes it. (; exec(0x73687574646f776e)–)

Most SQL injections are made through the POST body sent information, but they can be found in the GET parameters as well. In the categorize_attack() function we match a regular expression against the POST and GET parameters to see if it contains SQL specific tokens. If it does, each parameter is sent to the categorize_sql_injection() function. The function tries to categorize the injection in 2 steps:

1. A file containing regular expressions for each category is parsed and the injection is matched against each one. We get a value between 0-6, 0 for no match and 1-6 for the categories.
2. The Lex&Yacc parser is called to parse the injection, this will also return a value between 0-6.

. Both steps are used for now due to the incompleteness of the Lex&Yacc parser, in the future the first step will be removed.

```
if (my_regex("...",input_buffer))
{
AttackType = 1;//sql inj
struct post_body pb = break_post_body(input_buffer);
for (i=0; i<pb.nr; i++)
if (my_regex("..",pb.values[i]))
{
SpecificAttackType = categorize_sql_injection(pb.values[i]);
}
}
```

The value is returned by the categorize_sql_injection() function and becomes the value of the global variable SpecificAttackType. In the case of values greater then 4, the attack is considered too dangerous and the query isn't allowed to reach the content generator (StopHim variable becomes 1).

3.6.2 User/Password guessing

User/Password guessing is a type of brute force attack. Essentially the attacker tries to guess a user in a database by repeated tries of various combinations. He might try to guess the user first and then the password or might try both at once, or knowing the user from a SQL injection would focus just

on the password.

We're focusing on user and password as for anything else we have the general brute force attack type treated.

First, we check and see if the post body contains email, username and/or password keys. To help with this attack the logs table contains the user_post and pass_post fields, in which we log each time the POST values that were sent for user and password.

The get_last_guessings() function calls a query that checks for the last logs on this IP address that has values in these 2 fields. The number of such requests in the last 5 minutes is returned. If this number is greater then 5 it means this may be a brute force attack, if the number keeps increasing in a small timespan then for sure this is an attack. The AttackType is set and SpecificAttackType is set 1 for user/email, 2 for password and 3 for both.

```
struct post_body pb = break_post_body(input_buffer);
int ok_user_email=0,ok_pass=0;
for (int i=0; i<pb.nr; i++)
{
if (strcmp(pb.keys[i],"email")==0)
ok_user_email=1;
if (strcmp(pb.keys[i],"username")==0)
ok_user_email=1;
if (strcmp(pb.keys[i],"password")==0)
ok_pass=1;
}
if (ok_user_email==1 || ok_pass==1)
{
int count = get_last_guessings(r);
log_nr(count);
if (count>2)
{
AttackType = 2;//brute force guessing
if (ok_user_email==1 && ok_pass==1)
SpecificAttackType = 3;
else if (ok_user_email==1)
SpecificAttackType = 1;
else
SpecificAttackType = 2;
if (count>100)
StopHim=1;
}
}
```

3.6.3 Cookie meddling

Different Attacks on cookies exist, all involve meddling with it in one way or another. Session hijacking is one of the most known and tried attacks, and because most sessions have a connection with a cookie, if no other protection is done the attack is quite easy.

In general a PHP session will have a PHPSESSID name and a hashed value. An attacker may try to brute force this value until he gets a correct value.

But even more dangerous attacks exist based on this. If a cookie isn't encrypted and it remembers things like user id, then simply changing it might offer the attacker a way to log in as a different user.

We address this kind of attack in the following way. Looking at the output headers from the past we see which cookies were set and with what values. Then, looking at the current input headers we search for these cookies. If the cookie values are different then this is clearly an attack.

```
struct MCookie setcookie = get_last_set_cookie(f->r);
if (setcookie.key!=NULL)
{
char headers_in[10000];
printtable(f->r,f->r->headers_in,headers_in);
if (my_regex(setcookie.key,headers_in))
{
char * cookie_val = apr_pcalloc(...);
sprintf(cookie_val,"%s=%s",setcookie.key,setcookie.value);
if (!my_regex(cookie_val,headers_in)) //we have a changed cookie
{
AttackType = 3;
SpecificAttackType = 1;
ok=1;
}
}
}
```

3.6.4 XSS

Cross-Site Scripting (XSS) attacks are a type of injection, in which malicious scripts are injected into otherwise benign and trusted web sites. XSS attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are quite widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it.

This is another example of how a simple attack, which could be avoided quite easily, can make enormous damage. An attacker could inject a redirect

script to his website and also send with it all the user's information. This will expose user information and also render the page useless.

Testing against XSS follows a very similar pattern to testing against SQL injections. Only the classification is of course different.

A regex with parts that are found in XSS is tested against the post body. After that each parameter is classified inside categorize_xss_injection(). This matches a few regular expressions against the string and after that the Lex&Yacc interpreter for XSS is called.

Once a injection succeeded almost anything can be done, since there is an entire language at the disposal of the attacker. So trying to classify XSS attacks isn't that realistic. Though an effort was made to recognize the most common XSS attacks:

1. redirect - by using window.location
2. alert/confirm/prompt
3. DOM alteration
4. sending info with AJAX
5. spamming - creating loops with never-ending messages

```
 int categorize_xss_injection(char *s) {
FILE * s_patterns = fopen("/licenta/xss_injection_patterns","r");
char c;
char pattern[1000];
while ((c = fgetc(s_patterns)) != EOF) {
fgetc(s_patterns);
fgets(pattern, 1000, s_patterns);
pattern[strlen(pattern)-1]=0;


if (my_regex(pattern,s))
{
fclose(s_patterns);
return c-'0';
}
}
fclose(s_patterns);
return -1;
}
```

3.6.5 Buffer Overflow

We treat the buffer overflow attack by checking the length of the GET and POST parameters. Example of GET check:

```
 if (strlen(r->args)>256)
{
AttackType = 5; //BUFFER OVERFLOW BASIC ATTACK
SpecificAttackType = 1;
}
```

3.6.6 Canonicalization

To detect canonicalization attacks the GET arguments and the url are matched against "../" type strings, and different encoding values of these. AttackType is set to 6 and logged.

Some specific canonicalization attacks are listened to that are more dangerous and that require stopping the request. These involve strings like "/etc/passwd", "/etc", "/bin"... pretty much anything that tries to access UNIX system files.

3.6.7 Command execution

This is detected by looking in the GET and POST parameters for general UNIX command strings or combinations of "./" and words.

Besides that, different encoding characters are searched for, because many such attacks use alternate encoding to mask or in some cases (%0a) execute commands.

3.6.8 Brute force and profiling

Brute force attacks are essentially repeated requests in small time spans. So the application looks for exactly that. Comparing the findings of get_last_requests() with certain numbers gives us a small categorization: possible brute force, certain brute force, too many requests. The last one means that this may turn into a buffer overflow type attack, in the sense that the hacker is trying to occupy all system resources by overloading the server with requests.

Apache handles this quite well, the application will only log and stop the request from reaching the content generator at some point.

Besides this, profiling can be seen inside a brute force attack, by checking to see if the URL and GET parameters of the requests are different between them.

```
int count  = get_last_requests(f->r);
//in last 5 minutes
if (count>100)
{
AttackType = 8;
SpecificAttackType = 1;
```

```
if (count>500)
SpecificAttackType = 2;
if (count>1000)
SpecificAttackType = 3;
}
int count2 = get_last_profiling_requests(f->r);
if (count2*100/count>70)
{
SpecificAttackType = 4;//profiling
}
```

4

Use Cases

In this chapter we present examples for the different treated attacks. We will first explain how a general attack is made, then profile the site and at the same time create a brute force attack, then explain SQL injections more in depth with a complete scenario, then we'll move to user/pass guessing, cookie meddling, XSS, buffer overflow, canonicalization and command execution .

All explained scenarios are done on the test site, which has been constructed specifically to have these security holes.

4.1 Anatomy of an attack

Contrary to beliefs, attacks can basically be made on any browser, without the use of any tools. Of course tools are used for many things like profiling a site, but the most known attacks are the most dangerous because they don't need anything complicated. An attack follows a certain general path that we will describe below. As we previously discussed each attack is basically contained in one request.

- The security hole is found.
- Malicious text is sent in that part of the site.
- The request gets to the server and gets passed to the content generator.
- The content generator doesn't verify the request body and uses it.
- Attack takes place and in most cases results are visible by the hacker directly in the response.

4.1.1 Profiling

The first step a hacker does is profiling the web application he intends to attack. This is done to find initial security holes and to map the structure of the site. There are many profiling tools available, one of the most popular and complete one is "Burp Suite" which we'll use in this example. Besides other functions, Burp Suite offers a tool called Spider to profile a site.

Fig. 4.1. show the results of the profiling. The repeated queries were also recognized by the honeypot as a brute-force profiling attack and will be displayed as such in the admin panel.

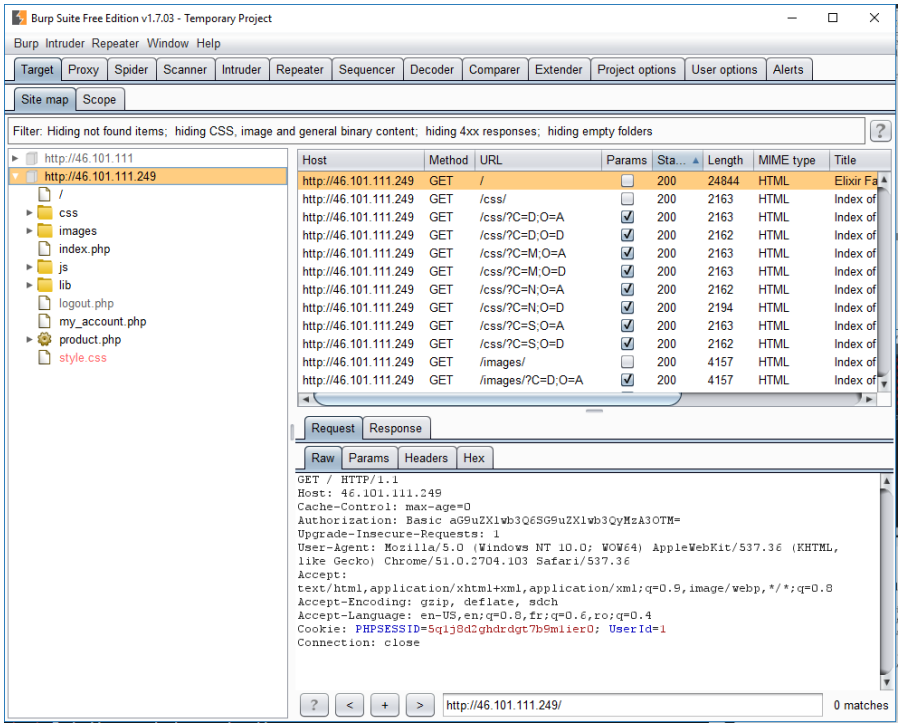

Fig. 4.1. Burp Suite Spider Example and Results

4.2 Sql injection attack

We will next describe a scenario in which an attacker can render the site useless, at the same time trying to categorize the attacks. Let's say that in the forgot password section the user is supposed to input his email and based on it he will get a mail with a link where he can edit his password. This situation exists in our test site, and the PHP code that handles the query for this is represented below with and without the security switch:

```
if (be_secure('BE_SECURE_LOGIN')==1)
{
$results = mysqli_query($conn,"SELECT * FROM users
WHERE email='".mysqli_escape_string($conn,$_POST['email'])."';");
}
else
$results = mysqli_query($conn,"SELECT * FROM users
WHERE email='".$_POST['email']."';");
```

Simply using mysqli_escape_string() renders the SQL injection mute, but let's say the input isn't parsed and it is used exactly as is:

Fig. 4.2. Forgot password section of the testing site

We'll start by probing the input to see if the possibility to attack exists. Usually this is done by inputting something like "*text' OR 1=1*". In our case the response will be "An email has been sent to ... with instructions for new password". This classifies as a tautology attack.

We might also try an logically incorrect query to see the result. So we'll send " "". The site doesn't break, because the mysql error is treated, and the response is "username not registered".

We can try next a UNION type attack to find what tables exist. Knowing that 'users' for example is a commonly used name for a table, we can send "*x' UNION SELECT * FROM users WHERE 'x'='x*".

This transforms the query into: "*SELECT * FROM users WHERE username='x' UNION SELECT * FROM users WHERE 'x'='x';*". The result is again "An email has been sent to ... with instructions for new password", which means the query was successful and that means the users table exists. We can now use this to try and find more info about this table or find other tables.

We can now try a piggy-backed query to modify or insert something in the database. At this point the honeypot will actually stop the request but for this example that functionality is deactivated. Also, since we're using mysqli in our test site, this limits the execution to 1 query, so for this example we're using mysqli_multi_query instead of mysqli_query.

Sending "*x'; DELETE FROM users WHERE 'x'='x*" now would delete the whole table. (thanks to the cloning functionality the actual database wasn't affected, just this attacker's clone, and he'll see the effects of that in reality)

Going even further in some cases a procedure attack could be done like "*x'; SHUTDOWN;*". In our case, the user has permissions only for his clone so this won't work. Fig. 4.3. shows a part of the log table (edited to remove header information) in the HoneyPot Admin for our last attacks.

2016-06-19 19:18:55	Piggy backed query	/signup.php	username=x'; DELETE FROM users WHERE 'x'='x&forgot_pass=1
2016-06-19 19:13:10	Union query	/signup.php	username=x' UNION SELECT * FROM users WHERE 'x'='x&forgot_pass=1
2016-06-19 18:58:35	Illegal/Logically incorrect query	/signup.php	username=''&forgot_pass=1
2016-06-19 18:57:16	Tautology	/signup.php	username=x' or 'x'='x&forgot_pass=1

Fig. 4.3. SQL attacks logs

4.3 User/Password guessing

User and password guessing is usually done by trying many different values for the respective fields. As our generated data has intentionally been composed using the 10000 most used users and passwords we will exemplify a simplified version of such an attack.

A small tool in JavaScript has been constructed for this purpose. It takes a list of users and passwords and tries the brute-force attack by trying all users first, when a user is found it tries all the passwords in combination with that user.

An attack is represented in Fig. 4.4. We can see that the user "user" is found and the password for it is "aboveground". Of course this is a small example and succeeded because we already knew what username and password to put in our list. A real attack contains from hundreds to thousands of such requests.

The Admin logs for the attack can be seen in Fig. 4.5.

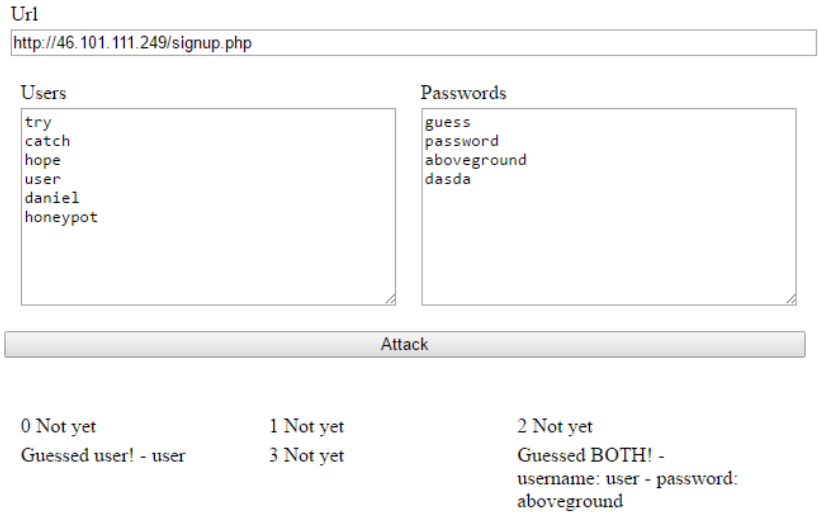

Fig. 4.4. Guessing Tool example

Date	Specific Attack Type	Uri	Request Body
2016-06-19 21:20:59	USer and Pass	/signup.php	username=try&password=guess
2016-06-19 21:20:59	USer and Pass	/signup.php	username=catch&password=guess
2016-06-19 21:20:59	USer and Pass	/signup.php	username=hope&password=guess
2016-06-19 21:20:59	USer and Pass	/signup.php	username=user&password=guess
2016-06-19 21:20:59	USer and Pass	/signup.php	username=user&password=password
2016-06-19 21:20:59	USer and Pass	/signup.php	username=user&password=aboveground

Fig. 4.5. Guessing Tool Admin Log

4.4 User Cookie Hijacking

To provide an example for the cookie meddling attack we will use the "remember me" function of the log-in section of our test site. That creates a non-encoded cookie with the user id, a common mistake in many sites.

Therefore, in logging in as 'user' we now have the cookie "*UserId=1*". If we now change that, as the site is unprotected we may login as another user. This is best tested on the my_account.php page.

Changing the cookie can be done even with a JavaScript call, or a simple tool like the "Web developer toolbar". While currently the my_account.php page shows "First Name: Quinn" after changing the UserId cookie to 14, the page shows "First Name: Maggy".

As we can see this can be a very dangerous attack, and can be done in just one simple request. The attack is caught and shown in the admin panel as well, Fig. 4.6.

Specific Attack Type	Uri	In Headers
Cookie changed	/my_account.php	http://46.101.111.249/my_account.php Accept-Encoding: gzip, deflate, sdch Accept-Language: en-US,en;q=0.8,fr;q=0.6,ro;q=0.4 Cookie: PHPSESSID=5q1j8d2ghdrdgt7b9m1ier0; UserId=14 If-None-Match: "1c44-

Fig. 4.6. Cookie Attack log

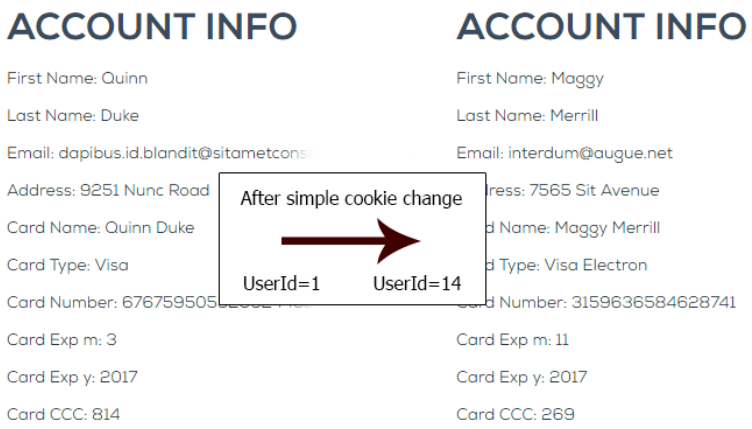

Fig. 4.7. Cookie Attack Result

4.5 XSS Attack

Similar to SQL injections, XSS attacks are quite diverse both in spectrum and in the damage they can do. We will present 2 attacks, both quite simple and yet destructive.

To do such an attack all you need is a place where the input that you send will be then displayed to other users. In our test site, this is the comments section for a product. Again, similar to SQL injections, these attacks could be easily prevented by verifying the input.

An attacker would first do a normal comment to see if it gets directly displayed or additional permission is needed.

First we will make a XSS redirect attack. Meaning in the input we'll insert *<script>window.location="http://google.ro";</script>* like in Fig. 4.8.

Next time we refresh the page, the comment will be shown, which will

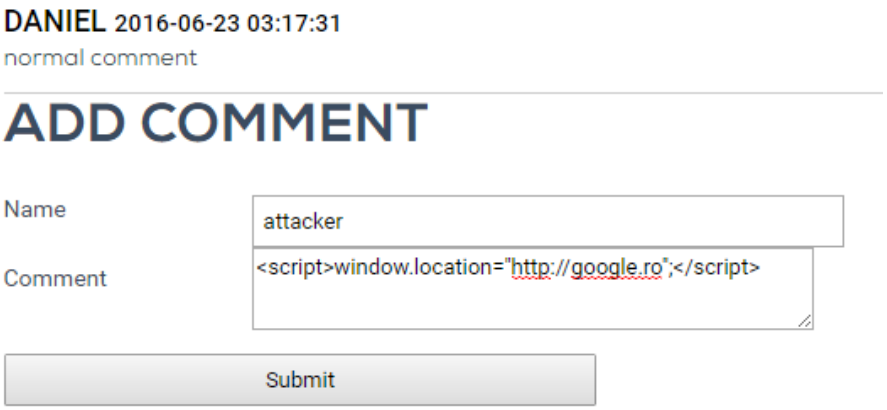

Fig. 4.8. XSS Redirect Attack

mean the HTML code will be created and the JavaScript code will be executed, redirecting us to google.ro. A simple attack but which transformed the whole page into a redirect page.

A more malicious attack would be to now use a script to take the cookie values and send them to us. So each user that will enter the page will automatically send their cookies to us. This, in combination with the cookie attack that we saw means access to the accounts of many users.

We can combine this with the redirect attack and insert in the input *<script>window.location="http://malicious.net?" + document.cookie.replace (';','&');</script>*

This will send the user to a malicious URL address and will put in the get parameters his cookie values. Better versions of this attack exist of course,

using AJAX for example to make the data transfer invisible to normal users, but for the sake of simplicity we demonstrated this short version.

4.6 Buffer overflow

Until recent a bug in Apache allowed hackers, through a simple buffer overflow attack to avoid having their requests logged by the server. Simply adding at least 255 more characters in the GET request did this. So an attack on the signup.php page could have been invisible by having the entire URL request be http://signup.php?attack= followed by 255 other characters.

Other simple buffer overflow attacks can be done in any input form that saves or modifies data. We know that usually fields like user name or password are saved in the database as VARCHAR types. This has a maximum length of 255, so by sending a greater number of characters we can make the query break.

This may not have a great effect on the application but it allows a user to see what messages the application throws when a query is broken.

4.7 Canonicalization

[5] a web application's security is always reduced to the lowest common denominator. Even a robust web server falls due to an insecurely written application. The biggest victims of canonicalization attacks are applications that use templates or parse files from the server. If the application does not limit the types of files that it is supposeed to vie, then files outside of the web document root are fair game.

An example for this kind of attack is the product page of the test site. In displaying the image of the product, the url format is:
http://46.101.111.249/product.php?id=2&product_image=Image00002.png
replacing the value of the product_image parameter with *../../../../etc/passwd* would make the server show the passwords file if the application isn't secure but has access to them.

The honeypot will stop such attempts as they classify as high risk attacks.

4.8 Command execution

[5] The newline character, %0a in its hexadecimal incarnation, is a useful character for arbitrary command execution. On Unix systems, less secure CGI scripts will interpret the newline character as an instruction to execute a new command.

An attack on our test site would look like this:
URL/product.php?id=1&%0a/bin/ls%0a
which if successful would of course display the content of the /bin folder.

4.9 Attacker story-line

An attacker(currently determined by an IP), usually does a number of attacks on a site. In our previous examples we have gone through all the types of the recognized attacks by the honeypot, and all these actions are now logged. A clear visual representation can be seen in the HoneyPot admin by going to "Attackers-> select "Visit" on any attacker".

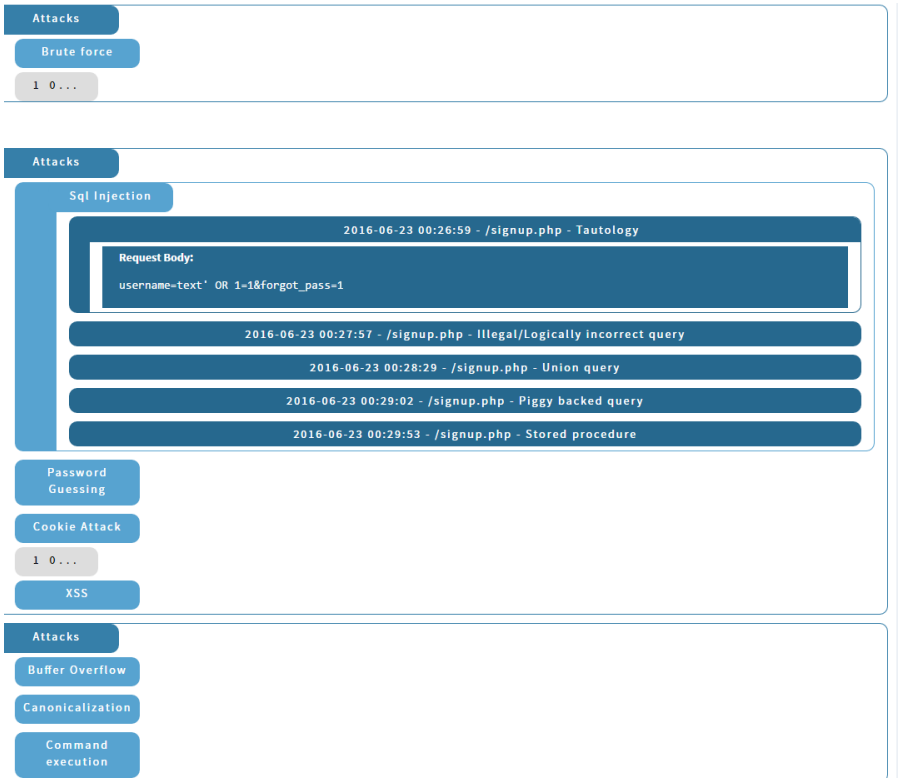

Fig. 4.9. Attacker story-line representation

We can see that each attack was caught, logged, and categorized. The logs are displayed ordered by time, and grouped by attack types. The SQL injection is opened in the image to show the details available on each attack group and attack.

5

Conclusions

In this paper we described an application meant to classify and detect web application attacks. We have discussed the different types of web application attacks, we have discovered the power an Apache module can have in analyzing every aspect of an attack, in it's every stage and we have made an effort in treating some of the more general attacks (which, as it was demonstrated, still exist and through their simplicity can be some of the most dangerous).

The open-source nature of the honeypot community has already been used, in verifying attacker IP's, but this is just a start, as honeypots should communicate, contribute and learn from each other.

Security is a property formed based on the different aspects of an application, and the different levels of filtering a request goes through. There is no 100% secure system, and new security breaches will always appear. Therefore the best we can do is keep up with them, test and detect them before malicious hackers. This author can only hope that this application will go on in being a helping hand in this process.

The current state of the project is still an early one, the focus of this thesis was to build a structure around the Apache module, and construct a honeypot system.

5.1 Moving Forward

Many ideas and improvements are planned in the future of the project, besides a better structure and all-around bug elimination. Here are a few:

- Implementation and use of a neural network for attack classification and future attack learning.
- Much more in-depth exploration of each type of know attacks.
- Active contribution to the honeypot communities.
- Establish a protocol through which honeypots can communicate findings and assist one another

References

1. Paco Hope, Ben Walther: Web Security Testing Cookbook, Systematic Techniques to Find Problems Fast O'Reilly, 2008
2. Nick Kew: The Apache Modules Book, Application Development with Apache Aprentice Hall, 2007
3. M. E. Lesk and E. Schmidt http://dinosaur.compilertools.net/
4. Stephen C. Johnson http://dinosaur.compilertools.net/
5. Joel Scambray, Vincent Liu, Caleb Sima: Hacking Exposed Web Applications 3, Web application security secrets & solutions Mc Graw Hill, 2010
6. SANS - What is a Honeypot? https://www.sans.org/security-resources/idfaq/what-is-a-honeypot/1/9
7. HoneyNet Project https://www.honeynet.org/about
8. Project Honey Pot https://www.projecthoneypot.org/about_us.php
9. Apache2 documentation for structures and functions https://ci.apache.org/projects/httpd/trunk/doxygen/structrequest__rec.html
10. httpd - Apache Hypertext Transfer Protocol Server https://httpd.apache.org/docs/current/programs/httpd.html
11. SQL definition of What IS http://searchsoftwarequality.techtarget.com/definition/SQL-injection
12. Glastopf https://github.com/mushorg/glastopf
13. Specter - Intrusion Detection Systematic http://www.specter.com/default50.htm
14. KFSensor http://www.keyfocus.net/kfsensor/
15. HoneyPoint Security Server http://microsolved.com/HoneyPoint-server.html
16. Honeyd http://www.honeyd.org/
17. OWASP https://www.owasp.org/index.php/Main_Page
18. OWASP WebScarab Project https://www.owasp.org/index.php/Category:OWASP_WebScarab_Project
19. Burp Suite https://portswigger.net/burp/
20. Generate Data .com http://www.generatedata.com/
21. The Apache HTTP Server Project https://httpd.apache.org/