# Cyber Fast Track:

## Web Application Honeypot

## Final Report

Author: Honeynet Project
Date: July 27, 2012

# Table of Contents

# Introduction

This document describes a web application honeypot developed by the Honeynet Project, a 501c3 non-profit security research organization, dedicated to investigating the latest attacks and developing open source security tools to improve Internet security. [Honeynet Project, nd] The Web Application Honeynet (WAH) project, described in this document, was funded as part of a DARPA Cyber Fast Track project to address a need in the web application security domain.

# Background

A Honeypot is "an information system resource whose value lies in unauthorized or illicit use of that resource" (Spitzner[1]). Honeypots come in many varieties, ranging from high interaction honeypots (HIH) in which real systems (such as operating systems, servers, and applications) are instrumented and deployed, to low interaction honeypots (LIH), which generally consist of emulated services that mimic some IT services or systems (e.g., a LIH that acts, at least superficially, like an IIS webserver). Another axis of differentiation is the range from server honeypots (i.e., honeypots that provides services that an attacker can connect to and interact with, such as a webserver), to client honeypots, which act as clients connecting to external systems (e.g., an instrumented web browser which connects to web sites in order to determine if they include malicious content).

Web applications are an increasingly visible and important part of the enterprise IT infrastructure, and we are seeing a similar increase in the amount of attacks against them. This project provides insight to the defensive community to help them understand the extent of the web application attack surface, and specifically what their adversaries are targeting and how they are doing so. This allows defenders to identify and secure potential vulnerabilities much earlier (potentially before they have been applied against real web applications at a particular organization, as opposed to after an attack has been successful).

## Web Application Honeypots

A web application honeypot (WAH) is a basic web server with an attack surface. This attack surface is the public HTML content which is indexed by search engines. It contains links to files with known vulnerabilities. The real vulnerability is not present but the web server advertises its existence and thereby attracts the adversaries. In order to handle an attack properly we need to classify the request. This is almost identical to what the web application firewalls are trying to achieve except that they are prone to extensive false negative as they have to deal with classifying a lot of legitimate traffic. The honeypot, by contrast, should see little legitimate traffic, which dramatically simplifies this classification process.
On the web application Honeypot, request handlers are responsible for classifying and handling each incoming request. We call them emulators as they emulate a vulnerability, and are responsible for creating the appropriate responses that leads the adversary to assume that he was

[1] Lance Spitzner,

successful. For example if the attacker tries to exploit a SQL injection vulnerability which discloses information from the database, our classifier should recognize the SQL injection and the emulator should return the error message or a page indicating to the attacker the attack was successful (e.g. listing the content of a database table).
Web Application Honeypots also include reporting capabilities which allow the defender (i.e., the administrator or researcher that deployed the WAH) to analyze and share the collected data.

Previously developed web application honeypots (such as HIHAT[2], Google Hack Honeypot[3], and DShield Web App Honeypot[4]) are only able to handle attacks that target the specific set of known vulnerabilities emulated by the honeypot. As a result, these current approaches are quite limited in their ability to respond to multi-stage attacks (e.g., SQL injection attacks which commonly start with a test to determine if a specific vulnerability is actually present in the targeted web application, followed by a exploitation of the vulnerability if it is seen to exist) and as such can only provide an incomplete picture of the tactics and motivations of the adversary.

In the web application domain we have observed many attacks which rely on the successful completion of multiple requests in order to send the final, and often the most interesting, payload. Previous web-application honeypots lack the ability to interact appropriately with the attacker throughout this entire chain, and as such were only able to provide very limited insight into the exploitation path, the post-exploitation behavior, and the motivation of the adversaries. Our vulnerability class emulation approach allows a much higher likelihood of responding correctly to a wider range of (particularly multi-stage) attacks, which will lead to a complete picture of the current and emerging web-application attacks.

Instead of manually extending our surface by implementing specific vulnerabilities as they become known, as in previous approaches, our method provides a dynamically expanding attack surface, which, in turn, attracts more attackers and exposes the honeypot to a higher number of interesting events (e.g., novel attacks).

## SQL Injection Background

"A SQL injection attack consists of insertion or 'injection' of a SQL query via the input data from the client to the application" [OWASP[5]]. Web applications commonly rely on databases to store the data required for their operations, including authentication and the dynamic generation of content (e.g., web pages). The interaction between the web application and the database generally involves the execution of one or more SQL statements (e.g., SELECT statements to retrieve data from the database, or INSERT to store and UPDATE statements to modify data in the database). These SQL statements are often constructed in part by using values supplied by the user (e.g., a username supplied by the user during an authentication operation, or the text of a comment supplied during the posting of a message to a page). As a result, it can be possible for an adversary to choose their input in such a way that it results in the construction of a SQL

[2] http://hihat.sourceforge.net/
[3] http://ghh.sourceforge.net/
[4] http://code.google.com/p/webhoneypot/
[5] https://www.owasp.org/index.php/SQL_Injection

statement that performs an operation that was not intended by the web application developer, and it is this manipulation of SQL statements that is known as SQL injection.

A successful exploitation of a SQL injection vulnerability may allow the adversary to execute unintended, or even arbitrary, SQL commands on the database (e.g., update or delete data, or retrieve data that should not be exposed to the attacker). In some cases, it is even possible to utilize such attacks to perform operations directly on the underlying operating system.

**Types of SQL Injections**

There are several methods to inject the malicious query including the following:

- User input in web forms
- Modified cookie fields
- Server variables like the HTTP headers
- Second order injection (i.e., stored and reused queries or data)

The attacker can manipulate this input and, if unchecked, the input will be passed to the database via the application's SQL commands.

The injection techniques can be separated in two main types:

- Classic injection: This type of attack modifies the intent of the SQL statement through user input to solicit a response indicating the attack has been successful. Modification of the SQL statement happens through user supplied input that is used by the web application in the construction of the SQL statement. Examples include UNION/piggy-backed queries, extending the regular query, obfuscation using various kinds of encoding and stored procedures. An example using UNION is shown below:

  ```
  SELECT header, txt FROM news WHERE header LIKE %"% UNION ALL SELECT name, pass FROM members; --'%;
  ```

  In this example, 'SELECT header, txt FROM news WHERE header LIKE %'INPUT'%' is the original query, '"% UNION ALL SELECT name, pass FROM members; --" is the part injected by the attacker. The resulting query would disclose the names and passwords stored in the database.

- Inference injection: This type of attack is behavior-based, either by conditional responses or time based. Since the adversary doesn't get data as feedback, this attack type is called 'blind injection'. Related SQL commands are SLEEP, BENCHMARK and 'waitfor delay'.

**Inference (blind) Error Based Example**

An interesting example combining error based injection with inference (blind) injection is demonstrated by the following query:

```
IF ((SELECT user) = 'sa' OR (SELECT user) = 'dbo') SELECT 1 ELSE SELECT 1/0
```

In the case that the current user is ‘sa’ or ‘dbo’ (similar in a database context to root on Linux or Administrator on Windows) the query will return 1, and if the current user is not ‘sa’ or ‘dbo’ the query result will be a ‘divide by zero’ error resulting from 1/0. If these query results are then reflected on the web page returned to the adversary that submitted the query, they can determine the user account being used by the web application.

**Multistage SQL Injection Attacks**

Attack types like Remote File Inclusion or Local File Inclusion are usually very effective attacks in terms of the amount of requests needed. A successful SQL injection needs much more probing and brute-forcing before an attacker is able to form a request that will successfully compromise the system.

Let’s assume our target path and parameter are the following:

```
http://honey.pot/path/file.php?parameter=
```

The first stage is usually an error based injection which is evident by the addition of a malformed parameter as shown below:

```
http://honey.pot/path/file.php?parameter=7′
```

If the injection was successful, the web application should return an error message similar to the following:

> “You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version...”

This would be the first evidence to the attacker, that the web application is vulnerable to SQL injection.

From this point, the attacker can start to retrieve as much information as possible about the database and its structure. An important piece of this puzzle is the number of columns in the current table. To get this piece of information, we can leverage the ORDER BY feature as follows:

```
http://honey.pot/path/file.php?parameter=7 ORDER BY 1/*
```
(/* truncates the SQL commands after the injection)

The attacker can then keep increasing the order number until another error is received:

> “Unknown column ’4′ in ‘order clause’”

From this message, and the fact that it was not encountered with an ORDER BY 3, the attacker knows that there must be three columns in the table. This information can be used in the next injection:

```
http://honey.pot/path/file.php?parameter=7 union all select 1, @@version, 3 /*
```

This should return a version number `'5.0.45'`

So we see that until there is a successful injection, there can be a significantly large number of requests. (The above demonstration would be a best-case scenario). This means an attacker has to select his victims carefully (usually by looking at the version number of a specific web application he is targeting) and the honeypot on the other hand has to respond properly in order for the probing requests to continue.

# The Web Application Honeypot

The web application honeypot developed during the Cyber Fast Track project is a web server providing an attack surface that is able to extend itself. The honeypot classifies incoming requests and handles them accordingly. We developed a new handler especially for SQL injections to be able to respond properly to all the stages of an attack until we get the final payload (e.g. content injection, disclosure of data, etc.). The honeypot can be deployed to collect malicious requests and then analyze their origin, purpose, and the targeted web application. The collected data can be leveraged to protect web applications before they get compromised.

# Attack Surface

In the context of this WAH, there are two important concepts to define in order to understand how the WAH provides an interface with which adversaries can interact:

- *Dork*: A dork is the bait attracting the attacker. An adversary is looking for a vulnerable path in our application (e.g. /path/to/vulnerable.php), and we refer to this path as a *dork*. Adversaries find these dorks via search engines, which index them as part of their standard operations (e.g., crawling the web and following links).

- *Attack surface*: The attack surface is a HTML page containing a large amount of dorks. A search engine crawler will fetch this page and add the dorks to its index. Querying the search engine for the characteristic of a potentially vulnerable web application will return our honeypot dorks in the search results (probably among other results which point to real and vulnerable web applications).

## Attack Surface Generation

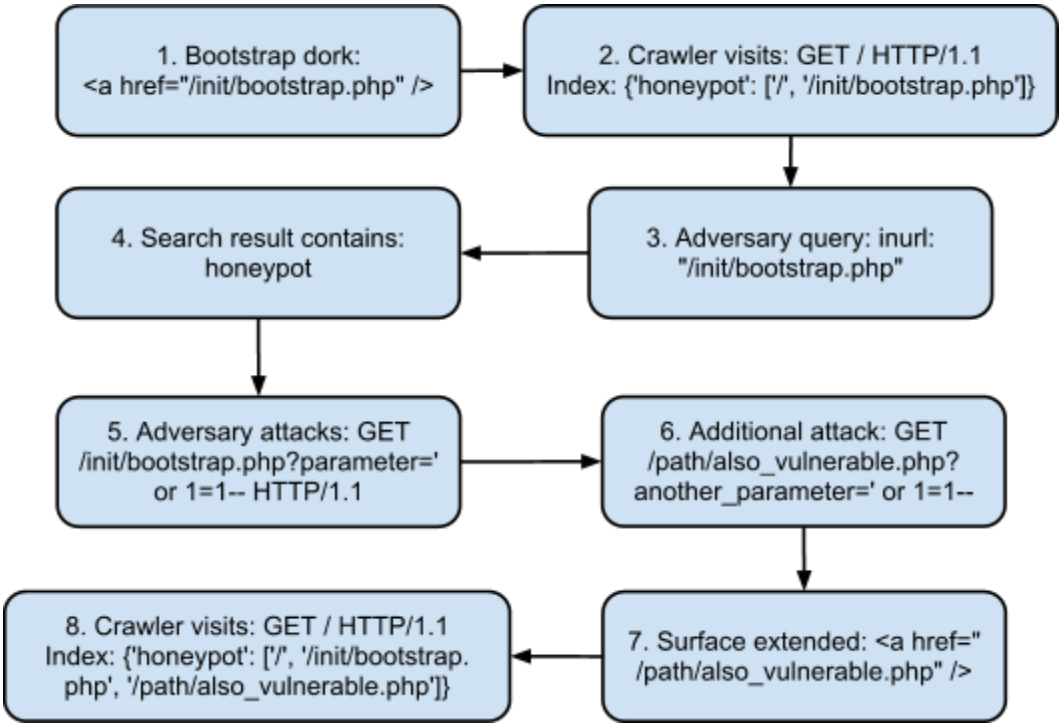
The dorks for the generation of the attack surface can be collected in two different ways. First we have existing data, previously collected events and external sources, also used by the adversaries to search for vulnerable web applications. Secondly we have the incoming requests. As many of those requests are probing for the existence of vulnerabilities, some of which we may not have seen previously, we can leverage them to increase the size of our attack surface.

**Dorks from existing and external data**

First we get the dorks from the bootstrap list file, which can be found at *modules/handlers/emulators/dork_list/dorks.txt*. The bootstrap list is a collection of commonly used dorks and previously collected attacks and from the sample honeypot event database (*db/events.bson*)

The dork information is used to fill the HTML template. As many search engines filter web pages which provide (from their point of view) useless data, we have to mask our attack surface with text from a corpus of suitable text. In the current case, we use randomly selected text from a freely available book, the text of which can be found at *modules/handlers/emulators/dork_list/pride.txt*

**Dorks from incoming Requests**




1. We start with an attack surface only holding a single link:
   `<a href="/init/bootstrap.php" />`
2. The first visit we get is from a search engine crawler: `GET / HTTP/1.1`
   The crawler finds the link and adds it to his index: *{'honeypot': ['/', 'init/bootstrap.php']}*
3. An adversary found a vulnerability in this PHP file and uses a search engine to find new victims using a query like:
   `inurl:"init/bootstrap.php"`
4. The search engine looks up the path and returns our honeypot in the search result.
5. The adversary now runs his attacking tool against the vulnerable PHP file on our honeypot

```
GET init/bootstrap.php?parameter=' or 1=1-- HTTP/1.1
```

6. The honeypot handles the attack. Attack tools are usually trying additional attack vectors, for example:
   ```
   GET path/also_vulnerable.php?another_parameter=' or 1=1--
   ```
7. So from this attack we see that there must be a vulnerability in *path/also_vulnerable.php* The attack surface expansion feature now extracts the path *path/also_vulnerable.php*, adds it to our database and in the next attack surface regeneration process, we will see a new link: `<a href="path/also_vulnerable.php" />`
8. Later we see the search engine crawler again (`GET / HTTP/1.1`) which results in the new link being added to the search engine index, which now includes: *{'honeypot': ['/', 'init/bootstrap.php', 'path/also_vulnerable.php']}* and another adversary looking for the new file will find our honeypot now.

This cycle continues to repeat as long as the web application honeypot is operating.

## Advanced Dork Selection

The advanced dork selection feature provides the possibility to select dorks based on a specific characteristic. Similar dorks are chosen using a k-means clustering algorithm. This gives us the possibility to focus on a specific attack types or web applications.

### Selecting Algorithm: K-Means Clustering

K-means clustering aims to partition a set of *n* observations into *k* clusters. The observations are partitioned based on the similarity of the selected features. The result are Voronoi cells, i.e., the similarity distance of the cluster members define a metric space.

With the observations $x_i$ and the clusters $S_i$ our metric is defined as:

$$\arg\min_{\mathbf{S}} \sum_{i=1}^{k} \sum_{\mathbf{x}_j \in S_i} \|\mathbf{x}_j - \boldsymbol{\mu}_i\|^2$$

Where $\mu_i$ is the mean of points in $S_i$.[6]

### Cluster Feature Selection

In order to partition the events into the clusters, we have to turn the event into a feature vector. We are using the clustering to get paths which are similar and most likely belong to the same web application. Wordpress for example uses ‘wp-’ as an identifier in their path/directory names. By creating a cluster with e.g. ‘wp-admin’ as feature, we get a selection of vulnerable paths related to Wordpress’ back-end.

As we are interested in the request path (e.g. */path/to/vulnerable.php*) we use the regular expression /\w+ which would return the tokens [‘/path’, ‘/to’, ‘/vulnerable’] (we are not interested in the file extension). The vectorizer[7] turns the raw input into a matrix of token counts.

[6] http://en.wikipedia.org/wiki/K-means_clustering#Description

### Example Cluster

The TimThumb vulnerability was a severe vulnerability in a PHP script used to scale images. As TimThumb was used in many Wordpress themes, providing lots of themes would very likely attract a large number of attacks targeting this specific vulnerability. We created clusters from all requests containing 'wp-' using the regular expression '.*?\/wp-.*' and selected the cluster with the features 'wp-content' and 'themes'. The result is a list of possible theme names and will provide a good bait for timthumb related attacks against web applications.

```
/wp-content/themes/wpclassifieds/functions/timthumb
/wp-content/themes/eGallery/timthumb.php
/wp-content/themes/ElegantEstate/timthumb.php
/wp-content/themes/wpclassifieds/modules/timthumb.php
/wp-content/themes/delight/scripts/timthumb.php
/wp-content/themes/portal/includes/timthumb.php
/wp-content/themes/wpclassifieds/scripts/timthumb
/wp-content/themes/wpclassifieds/library/timthumb
/wp-content/themes/SimplePress/timthumb.php
/wp-content/themes/headlines/
*truncated*
```

## Additional Dork Sources

To successfully find vulnerable web sites, adversaries need information about the existing vulnerabilities. IRC channels, bulletin boards and public vulnerability databases are the most common ways. http://exploit-db.com is one of those databases. Originally build up by Johnny Long as the Google Hack Database (GHDB) the exploit-db is a rich source for dorks (i.e. search terms used to find vulnerable web applications) and the requests needed to exploit a vulnerability. So adding the dork terms and the involved paths and files to our attack surface will very likely increase our attractiveness (i.e. the amount and diversity of attacks). The external source dork extraction is implemented as handlers. Every source needs a specific handler capable of reading the provided format.

### Dork extraction

To use the database, we took a dump of the web application related exploits, each of which are stored in separate files. To extract the dorks an exploit-db specific extraction handler was implemented. To get the paths and dorks we made use of regular expressions and keywords.

To extract the paths, we used the regular expression:

```
(http[s]?://[a-z]+?\.?[a-z]*)(/[^\s]+?\.[a-z]+)(\?)([^\s]+=[^\s]+)
```

The keyword 'sql' on the exploit-db collection returned 409 matches from a total of 2937 extracted paths from 8665 vulnerability reports.

The keyword 'dork' returned a total of 432 dorks that can be used as bait on our attack surface.

---

[7] http://scikit-learn.org/dev/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html

As many of the vulnerability reports are not following a standard format, extracting turned out to be not complete but successful enough to get additional attack vectors.

**Sample vulnerability report:**

> *osCommerce Online Merchant 2.2 RC2a RCE Exploit*
> Dork: "**Powered by osCommerce**"
> Request: POST /admin/file_manager.php/login.php?action=save
> Payload: filename=fly.php&file_contents=test<?php @eval(\$_POST[aifly]);?>

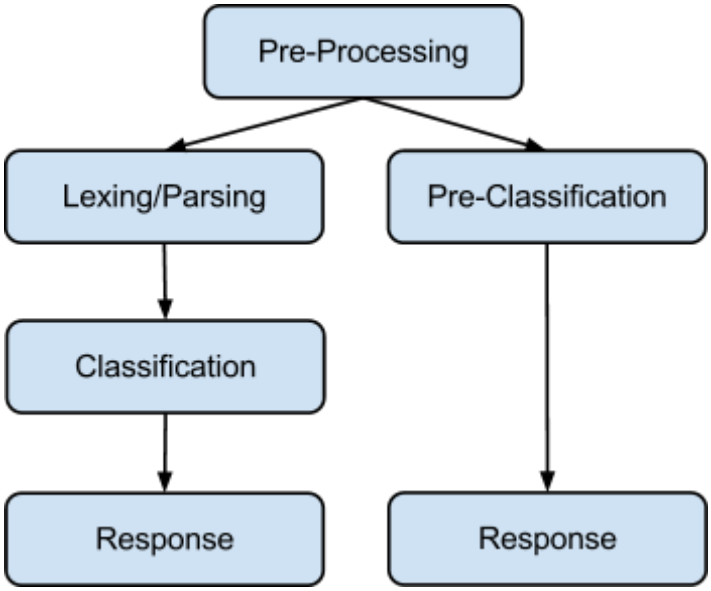
This request abuses a vulnerability in osCommerce by uploading a shell which allows execution of arbitrary commands on the compromised web server. Our extraction process would get the dork 'Powered by osCommerce' and add the link:

```
<a href="/admin/file_manager.php/login.php" />
```

to the attack surface.

# SQL Injection Handler

In order to properly respond to SQL injections and to provoke additional attacks, we developed an SQL injection attack type handler. The request handling was divided in two branches, one to handle SQL queries which are parsable and one for the non-parsable parts usually using pattern matching as shown in the figure below.




The SQL injection handler for parsable queries is based on lexical analysis and a parser processing the analysis result as shown in the following example:

Example query: `SELECT A FROM B`
Lexer output: `'121, 237, 80, 237, 122, 237, 80'`
Parser output:

```
(SELECT (SELECT_CORE (COLUMNS (ALIAS (COLUMN_EXPRESSION A)))
(FROM (ALIAS B))))
```

## Pre-Processing

In the pre-processing step, we de-obfuscate the query so the lexer is able to consume it. Additionally we are looking for functions which require computation from the database engine (e.g., CONCAT() which is used to concatenate the parameters). By pre-computing such statements we narrow down the list of possible matching queries in the classification stage.

## Pre-Classification

In the pre-classification step we are looking for queries which are not parsable. These queries are usually trying to break the existing query to provoke an error message. Patterns for this step are stored in modules/classification/sql_utils/patterns.xml. The pre-classifier compares the beginning

of the query with these patterns. Each identified pattern contains information on how to respond to the request.

## Lexing/Parsing

The lexer and parser is generated from the grammar definition in the following file:

*modules/classification/sql_utils/SQLite.g*

The prepared SQL query is now lexically analyzed and the tokens parsed. As some of the queries are meant to be appended to an existing one, we have to use some error recovery and brute force algorithms to properly understand it. For example the following SQL injection tries to inject their own logic into an existing query:

```
http://server.tld/login.php?user=admin&password=fubar' OR 1 = 1
```

If we just take the parameter password:

```
'fubar' OR 1 = 1'
```

our parser will fail as the SQL query is not complete. The parser expects something similar to the following:

```
SELECT user_id from users WHERE password == 'fubar' OR 1 = 1
```

In order to parse the injection properly, we have to add at least:

```
SELECT a WHERE b == '
```

to the query in order to parse it.

Lexical analysis turns the query in a series of tokens which generalizes the request and allows us to improve classification and reduce the query to easy to identify pieces. These tokens are defined in *SQLite.g* and listed in *modules/classification/sql_utils/SQLite.tokens*

Parsing turns the token sequence in a logical expression, allowing us to understand the query's purpose and goal.

## Classification

With the tokens and the parsed query tree, we are able to classify the request. The classification process consists of a token comparison step and a step where we compare the query against the rules defined in modules/classification/sql_utils/queries.xml.

Every supported rule has information about the content of the response. For example a request which is supposed to disclose table names should get a response containing such a list. The corresponding rules should provide such a list of table names.

## Response Generation

There are two different paths through the SQLi handling routine that result in a response:

### From pre-classification

With the result from the pre-classification, the attack payload and the response content defined in modules/classification/sql_utils/responses.xml we are able to generate the response for attacks trying to provoke an error message.

The following is an example entry in *responses.xml*:

```
<response>
    <id>mysql_error</id>
    <description>MySQL error message</description>
    <content>
             Invalid query: You have an error in your SQL syntax; check the
             manual that corresponds to your MySQL server version for the right
             syntax to use near 'PAYLOAD' at line 1
    </content>
</response>
```

In this case `'PAYLOAD'` will be replaced by the request's payload.

### From classification based on lexer and parser

With the result from the lexical analysis and token parser we are able to generate the responses from the most similar request defined in the query pattern file:

*modules/classification/sql_utils/queries.xml*

The following is an example entry in *queries.xml*:

Example entry in queries.xml:

```
<query>
      <database>
            Mysql
      </database>
      <query>
            SELECT user()
      </query>
      <parsed>
            (SELECT (SELECT_CORE (COLUMNS (ALIAS (FUNCTION_EXPRESSION
            user)))))
      </parsed>
      <tokens>121, 237, 80, 48, 50</tokens>
      <response>
            root@localhost
      </response>
</query>
```

In this case `'PAYLOAD'` will be replaced by the request's payload.

Using the parser tree we are also handle special cases like `SLEEP()` and `CONCAT()` in the emulation step. The trees logic features allows us to easily locate the function and its parameter.

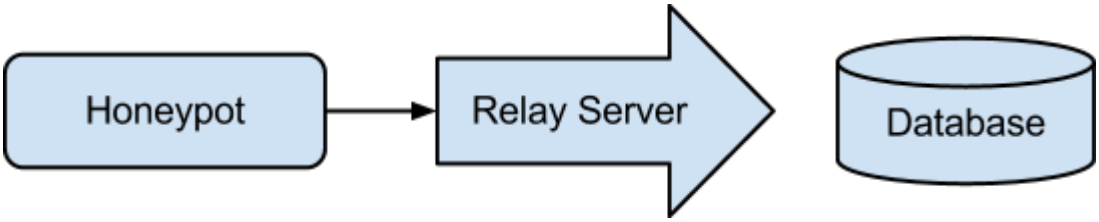
In the case that we are not able to parse the query or if the emulator is not able to generate a response, we fall back to an SQL error message containing the query.

# Deployment and Methodology

Since the completion of milestone 1 we had the honeypot deployed and accessible from a single public domain. Since the first working version of milestone 2, the deployed honeypot was updated so we had the finished SQL injection handler in place. After milestone 2 the honeypot was updated on a daily basis.

## Deployment Issues

During the period where we had the honeypot deployed, we had various issues causing the honeypot to stop collecting events or we lost connection to the remote database which also meant lost data.




As we collected the data in a machine separate from the honeypot to prevent compromised data in the possible event of a compromised machine running the honeypot, the connection between the honeypot, the relaying machine in between and the database was crucial to successfully record the events from the honeypot.
Since deployment of the draft versions of the SQL injection handler we also had some issues with the honeypots stability which caused loss of data. All stability issues got fixed with the completion of milestone 2. We mention it here to explain gaps and fluctuation of the collected data in the next sections.

## Data Quantity

In the preliminary testing period from 05/01/12 until 06/27/12, the total number of events collected was 657,541, an average of about 11,000 per day. During a whole day of sensor uptime we collected in average about 20,000 events. From this number we are able to estimate a total sensor downtime of about 50% of the time.
Extrapolating the total number of possibly collected events by assuming a uniform distribution, would result in more than one million events. In the result section we will see that our assumption is right for the overall amount of traffic; Mostly because of the high percentage of events caused by search engine crawlers. If we look at specific attack types, we see that they come in spikes. On some days we haven't seen a single main attack type. So the uniform distribution fits to the total amount of events, but for a single domain the amount of daily malicious events are unpredictable.

The majority of the events were classified as ‘unknown’. We assume they are created by search engine crawlers. ‘unknown’ events are non-malicious and usually caused by search engines, random visitors and attack tools pretending to be regular users. As we are expecting false negatives, especially for HTML injections, we took four random selections of hundred events which we manually examined revealed two false negatives (FN rate = 0.5%). Or in other words: We examined 400 events classified as ‘unknown’ aka non-malicious and found two which have been malicious (one HTML injection and a broken local file injection).
The large number of total events speaks for the success of our attack surface.

## SQL Attacks per Day

Selecting all SQL related requests turned out to be difficult as many probing requests are hard to be distinguished from non-malicious request caused by crawlers. Nonetheless looking at some of the attacks revealed an interesting, yet expected pattern. The gross of the request originated from a small number of IP addresses and occur in spikes.

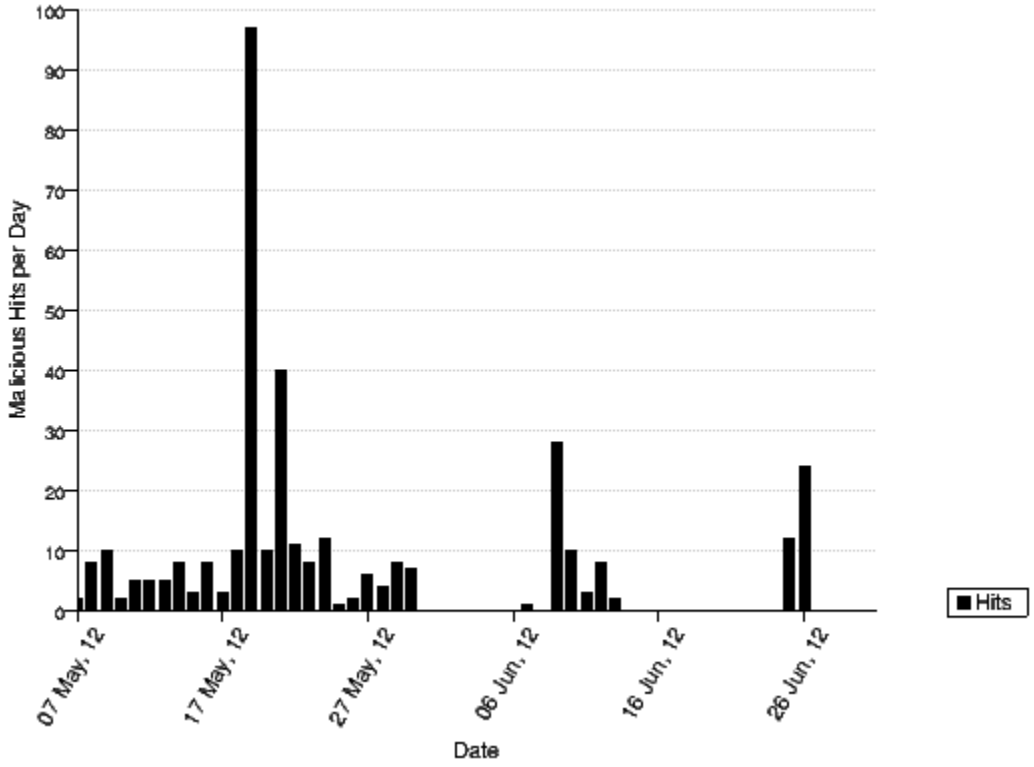



Figure 1, SQL attacks per day

The average 10 SQL injection attacks per day (Fig. 1) are still not very common compared to the 11 thousand total daily events. This could be explained by the short timespan between this report and the deployment of the new attack surface. It usually takes five to ten days until a link from the attack surface appears in the public search index and attacker can use them to find victims (or the honeypot).

# Results

During the development of the honeypot we deployed various revisions and collected large amounts of data. In the next sections we will give an overview over the collected attacks, point out the interesting findings.

## SQL Injection Attacks

We collected 363 SQL injection attacks which are using `'1=1'` to bypass a verification logic and `'UNION'` to add a malicious request to the existing one.

The injections originate from 69 different IP addresses. 4 IP addresses used very similar requests in the same time frame which could be an indication for an adversary using multiple machines to distribute the attacks load. We also found SQL injections originating from an IP address belonging to an search engine provider. Further investigation revealed that a path with parameters slipped to the attack surface during development, got picked up by the search engine crawler who tried to visit that link and thereby unintentionally attacking the honeypot.

SQL injection attacks usually start with a probe to determine if the victim is vulnerable. If the attacking tool is economical with his resource we will only see a second stage if we properly reply to the test requests.

**Investigation: 190.46.223.***

In this section, we investigate the series of requests coming from a specific IP address. The first request from 190.46.223.*, registered in Chile, was collected on 2012-06-12 17:02:13. The request targeted a vulnerability in the FrontPage extension in Microsoft ISS:

```
GET /*/_vti_pvt/d.aspx?id
```

Quite interesting the next request a minute later:

```
GET /favicon.ico
```

This is interesting because the favicon icon is usually only requested by browsers and we assume an automated attack tool.

Two and a half minutes after the first request we see the first real attack:

```
GET /*/_vti_pvt/d.aspx?id<script>alert('XSS')</script>
```

The adversary is now targeting directly the previously mentioned FrontPage vulnerability. He tries to inject HTML code which executes the JavaScript alert function.

After the first cross site scripting attack, we haven't seen any attacks for about an hour until 18:02:40 when he targeted another vulnerability. This time another cross site scripting vulnerability but this time in Oracle Siebel:

```
/loyalty_enu/start.swe/>"><script>alert('XSS')</script>
```

He continues then with a local file inclusion and more cross site scripting attacks:

18:03:10 Local File Inclusion:

```
/file/../../../../../../../../windows/system32/calc.exe
```

18:03:12 SAP Internet Transaction Server XSS vulnerability:

```
/scripts/wgate/!?~urlmime="><EEYEXSSTAG_URLMIME><img
```

18:03:16 Oracle GlassFish Enterprise Server Stored XSS:

```
/com_sun_webui_jsf/help/helpwindow.jsf?&windowTitle=Help+Window&helpFile=common
task.html'>\"><EEYE2011XSS>
```

The last one is another interesting finding. The injected code contains the string `'EEYE2011XSS'` which can be linked to a company called eEye Digital Security which sells a vulnerability scanner.

All the previously shown attacks haven't been successful as we don't support HTML injection or local file inclusions targeting the Windows file system.

Injection

```
/forums/index.php?act=Arcade&cat=1'
```

The attacker is provoking a SQL error message. The honeypot returns a HTTP status 200 as the request was successful but includes the SQL error message:

```
Invalid query: You have an error in your SQL syntax; check
the manual that corresponds to your MySQL server version
for the right syntax to use near '1' at line 1
```

Similar responses are used for every follow up error based SQL injections.

```
18:04:09 UBBThreads Input Validation Error in 'dosearch.php
     /dosearch.php?Name=1'
```

It first seemed like he is not interested or our response doesn't matches his expectation but the next couple of attacks are second stage attacks.

First the vulnerability probing as we have seen it before:

```
18:04:11 AntiBoard antiboard.php script SQL injection
     /antiboard/antiboard.php?thread_id='
```

And then the real attack at 18:04:12 trying to disclose the user database:

```
/antiboard/antiboard.php?thread_id=1%20UNION%20ALL%20select
%20*%20from%20users--&mode=threaded&sort_order=
```

The injected query looks after URL decoding like this:

```
1 UNION ALL select * from users
```

The adversary is trying to disclose the data stored in the table ‘users’. The honeypot response looks like this:

```
debian-sys-maint root testuser
```

After that we see another set of probing attacks:

```
/print.php?what=article&id='
/showproduct.php?product=1'
/showcats.php?cat=1'
```

Where the last one is followed by another information disclosing attack:

```
/showcats.php?cat=1+union+select+1,concat(username,0x3a,pas
sword),3,4+from+sblnk_admin
```

And:

```
/showcats.php?forumid=-
1%20union%20select%20ModName%20from%20modretor
```

The last two queries are very application specific and we are not able to respond with the proper data.

At 2012-06-12 18:04:14 we see a last request targeting a SQL injection vulnerability in BosDates

```
/calendar_download.php?calendar='
```

Overall this attacker was quite interesting. We have seen him targeting the easy vulnerabilities in the beginning (i.e. remote file inclusion, local file inclusion and cross site scripting) followed by the more complicated SQL injection attacks.

**Multi Stage Attack**

*Stage 1*
In this request series the attacker starts with a test to determine if the web application expects a character (here the question mark) or integer (in this case '5') input:

```
search/cgi.asx?StoreID=?+and+1=1
search/cgi.asx?StoreID=5+and+1=1
```

*Stage 2*
Providing an integer was successful and the adversary continues to determine how to concatenate his malicious part of the SQL injection:

```
search/cgi.asx?StoreID=5;+if+(1=1)+waitfor+delay+'00:00:09'--
search/cgi.asx?StoreID=5';+if+(1=1)+waitfor+delay+'00:00:09'--
search/cgi.asx?StoreID=5+and+if(1=1,BENCHMARK(14970800,MD5(0x41)))
search/cgi.asx?StoreID=5'+and+if(1=1,BENCHMARK(14970800,MD5(0x41)))+and+'x'
='x
```

'waitfor delay' and BENCHMARK are both time based blind SQL injection attacks.

**User credential disclosure**

*Stage 3*
If the probing request was successful, the adversary tries to disclose usernames and passwords. In this case the targeted web application seems to be Joomla (jos_users is the Joomla database holding the user accounts).

```
admin?mail=-
1+union+select+concat(username,0x3a,password)+from+jos_user
s--
```

**Multi Stage Example**
Here is an example for a multi-stage attack.

At 00:39:37 on 2012-06-26 we have seen the first probing attack from 4.30.110.178:

```
GET /template.php?mod='
```

We are not hundred percent sure, but tThe adversary is most likely targeting a SQL injection vulnerability in the PHPBB forum web application.

As his request returns the expected error message, he continues with a query to determine if the vulnerability provides the possibility to concatenate queries using the UNION command:

```
/template.php?mod=-1+union+select+0x6c6f67696e70776e7a--
```

The injected hexadecimal encoded string ‘0x6c6f67696e70776e7a’ translates to ‘loginpwnz’. If the UNION was successful, the response should contain the decoded string (i.e. the vulnerability provides concatenating using UNION and hex decoding).

To bruteforce the number of needed columns, the attacker keeps adding column names to the SELECT statement:

```
/template.php?mod=-1+union+select+0x6c6f67696e70776e7a--
/template.php?mod=-
1+union+select+0x6c6f67696e70776e7a,0x6c6f67696e70776e7a--
/template.php?mod=-
1+union+select+0x6c6f67696e70776e7a,0x6c6f67696e70776e7a,0x6c6f67696e70
776e7a--
/template.php?mod=-
1+union+select+0x6c6f67696e70776e7a,0x6c6f67696e70776e7a,0x6c6f67696e70
776e7a,0x6c6f67696e70776e7a—
```

# Conclusion

The modular design of the Web Application Honeypot provides a platform for easy extension of the request handling capabilities, logging features and feature testing. We used this for the development of the new handler and the improvements of the attack surface. Overall the new attack type handler is starting to process the SQL injection attack. As the new attack surface is only deployed since a couple of days before this report, there is no significant change in the type and amount of requests we see. This will very likely change as soon the new dorks are propagated through the search engines index and show up in the search results. The results from the lexer/parser are very promising, we are able to dissect the malicious request into the query logic which we are able to parse and understand. Currently some of the classification steps and response generation elements are based on patterns but with the experience from handling special cases like query concatenating using ‘CONCAT’ or the ‘SLEEP’ function used in blind SQL injection attacks which maps the parsed query directly to a logic implemented in Python, we have the foundation for a very flexible SQL injection handler. The attack surface general approach is successful and future data analysis will reveal if the new features, like data clustering for dork selection and external dork sources, will increase the amount of malicious requests per day.

# Appendix

## Installation

The honeypot is developed for Debian or Debian based Linux distributions. In this guide we assume a recent Ubuntu host system.

### Basic dependencies:

```
sudo apt-get install git subversion python2.7 python-openssl / python2.7-dev build-essential make sqlite3
```

### ANTLR Python runtime for the SQL parser and lexer:

```
cd /opt
sudo wget http://www.antlr.org/download/antlr-3.1.3.tar.gz
sudo tar xzf antlr-3.1.3.tar.gz
cd antlr-3.1.3/runtime/Python
sudo python2.7 setup.py install
```

### SKlearn

Dependencies:

```
sudo apt-get install python-dev python-numpy python-setuptools / python-numpy-dev python-scipy
libatlas-dev g++ git
```

SKLearn for dork clustering:

```
cd /opt
sudo git clone git://github.com/scikit-learn/scikit-learn.git
cd scikit-learn
sudo python2.7 setup.py install
```

Note: Warning messages about not finding certain files can be ignored.

### Mongo DB for dork and event logging:

```
sudo apt-get install mongodb python-pymongo
```

Import sample db for test cases:

```
cd /opt/honeypot/db
sudo mongorestore -d honeypot -c events events.bson
```

HTML parsing to test the dork page generation:

```
sudo apt-get install python-lxml python-beautifulsoup
```

### Web Server / Honeypot

evnet module for the web server:

```
cd /opt
sudo git clone git://github.com/rep/evnet.git
cd evnet/
sudo python2.7 setup.py install
```

Get the honeypot source from the Subversion repository:

```
cd /opt
sudo svn co svn://glastopf.org:9090/cft_glaspot honeypot
```

### PHP sandbox

APD[8] is a PHP extension which is needed to run the sandbox. The sandbox is overwriting critical PHP functions and APD provides this capability.

Dependencies:

```
sudo apt-get install php5-cli php5-dev subversion python2.7 \ build-essential make
```

Download the latest apd source and build it:

```
cd /opt
sudo svn co http://svn.php.net/repository/pecl/apd/trunk apd
cd apd/
sudo phpize
sudo ./configure && sudo make && sudo make install
```

Add the following lines to your /etc/php5/cli/php.ini (make sure the zend_extension path is correct):

```
zend_extension = /usr/lib/php5/20090626+lfs/apd.so
apd.dumpdir = /tmp/apd
apd.statement_tracing = 0
```

Test with # php5 --version, output should contain something like this:

*PHP 5.3.9-1 with Suhosin-Patch (cgi-fcgi)*
*Copyright (c) 1997-2012 The PHP Group*
*Zend Engine v2.3.0, Copyright (c) 1998-2012 Zend Technologies*
*with Advanced PHP Debugger (APD) v1.0.2-dev, , by George Schlossnagle*

Go to sandbox directory opt/honeypot/sandbox and create the apd_sandbox.php using:

```
cd /opt/honeypot/sandbox
sudo make
```

## Usage

### Configuration and running the Honeypot

Configure IP address and port for the listener in the file honeypot.cfg
To run the honeypot issue the following command in the installation root directory:

```
sudo python2.7 webserver.py
```

### Testing the Honeypot:

Run the test suit provided in the honeypot installation root directory:

```
sudo python2.7 test.py
```

A HTML formatted test report will be available in testing/reports

### Reading logs

[8] http://pecl.php.net/package/apd

A simple text trace log is created in log/honeypot.log, which can be read with
```
cat log/honeypot.log
```

A SQLite3 DB is created atdb/honeypot.db that contains the various events. It can be read by executing
```
sqlite3 db/honeypot.db
```

Issuing for example the following SQL statement:
```
SELECT * FROM events;
```
returns all logged events in the database.

## Logging

Basic honeypot status messages are logged to log/honeypot.log
Log entry format is:
%(asctime)s %(levelname)s %(message)s
Example log entry:
2012-05-18 08:57:20,213 INFO 127.0.0.1 GET /index.php

Detailed event log can be found in db/honeypot.db
Database structure:

| Field Name | Field Type | Description | Example |
|---|---|---|---|
| id | INT | Event id number | 1 |
| timestamp | TEXT | Event time | 2012-02-13 21:44:06 |
| source_addr | TEXT | Request source address | 66.249.72.X:35 |
| request | TEXT | Request path | /admin |
| module | TEXT | Classification result | unknown |
| filename | TEXT | Name of injected file | None |
| response | TEXT | The honeypots response | HTTP/1.1 200 OK ‘truncated’ |
| host | TEXT | The host requested by the source address | honeypot.tld |

By loading the database in db/honeypot.db (sqlite3 honeypot.db) with a SQLite client and issuing the query:
**SELECT** * **FROM events WHERE module** != 'unknown';
You are able to get a list of all the successful classified requests monitored by the honeypot.

## Extending

Extending the honeypot by a additional emulator needs two steps.

1. Add a rule to the /request.xml which matches the requests handled by your new module. The new modules file name has to match the module defined in the new classification rule.

2. Add a new module in the /modules/handlers/emulators directory. As a minimum requirement for the module you have to inherit your new module from the modules.handlers.BaseEmulator class and provide a function
handle(self, attack_event) to accept the request.

### Example module:

```
from modules.handlers import base_emulator

class NewHandler(base_emulator.BaseEmulator):
    def handle(self, attack_event):
        pass
```

### Example request.xml pattern entry:

```
<request>
    <id>X</id>
    <patternDescription>Example pattern</patternDescription>
    <patternString>/example</patternString>
    <module>example</module>
  </request>
```

## TESTING

The honeypots functionality was tested using the unit test methodology. Every core feature was covered by a separate test case.

### Sample Test Case Output

Demonstrating the lexical analysis module test case output if the test was successful.

| | |
|---|---|
| ***test_sqli_lexer***:<br>Objective: Tests the SQL injection lexer.<br>Input: 'SELECT A FROM B'<br>Expected Results: Query tokens 121, 237, 80, 237, 122, 237, 80<br>Notes: 121 matches the SELECT, 237 the three white spaces, 80 the column and table alias and 122 the FROM | **Test: pass**<br>Starting SQL injection Lexer test...<br>Sending request:/test.php?q=SELECT A FROM B<br>Return value:<br>Query tokens: '121, 237, 80, 237, 122, 237, 80'<br>equates our expectation. |

To test a new emulator, add the following code to the /testing/test_emulators:

```
def test_new_emulator(self):
    """Test description"""
    self.event.matched_pattern = "emulator name (file name)"
    emulator = request_handler.get_handler(self.event.matched_pattern)
    emulator.handle(self.event)
    self.assertEqual(self.event.response, "expected response")
```