Hacker Intelligence Initiative, Monthly Trend Report #4

Hacker Intelligence Summary Report – An Anatomy of a SQL Injection Attack

This month's report from Imperva's Hacker Intelligence Initiative (HII) focuses on the rise in SQL Injection (SQLi) attacks on the Web. Dominating headlines for the past year, SQLi has become a widely-known, even outside the circle of security professionals. And for good reason: SQL injection is probably the most expensive and costly attack since it is mainly used to steal data. Famous breaches, including Sony, Nokia, Heartland Payment Systems and even Lady Gaga's Web sites were compromised by hackers who used SQL injection to break-in to the application's backend database. LulzSec, the notorious hacktivist group, made SQLi a key part of their arsenal. This report details how prevalent SQL injection attacks have become, how attacks are executed and how hackers are innovating SQLi attacks to bypass security controls as well as increase potency. From 2005 through today, SQL injection has been responsible for 83% of successful hacking-related data breaches.[1]

By monitoring a set of 30 web applications over the last nine months, we found:

- **SQL Injection continues to be a very relevant attack**. Since July, the observed Web applications suffered on average 71 SQLi attempts an hour. Specific applications were occasionally under aggressive attacks and at their peak, were attacked 800-1300 times per hour.
- **Attackers increasingly bypass simple defenses**. Hackers are using new SQLi attack variants which allow the evasion of simple signature-based defense mechanisms.
- **Hackers use readily-available automated hacking tools**. While the attack techniques are constantly evolving, carrying out the attack does not necessarily require any particular hacking knowledge. Common attack tools include Sqlmap and Havij.
- **Attackers use compromised machines to disguise their identity as well as increase their attack power via automation**. To automate the process of attack, attackers use a distributed network of compromised hosts. These "zombies" are used in an interchangeable manner in order to defeat black-listing defense mechanisms.
- **About 41% of all SQLi attacks originated from just 10 hosts**. Again, we see a pattern where a small number of sources are responsible for a majority of attacks.

To better deal with the problem, enterprises should:

- **Detect SQL injection attack.** Using a combination of application layer knowledge (application profile) and a preconfigured database of attack vector formats. Detecting SQLi must normalize the inspected input to avoid evasion attempts.
- **Identify access patterns of automated tools**. In practice, SQLi attacks are mostly executed using automatic tools. Various mechanisms exist to detect usage of automatic clients, like rate-based policies and enforcement of valid client response to challenges.
- **Create and deploy a blacklist of hosts that initiated SQLi attacks**. This measure increases the ability to quickly identify and block attackers. Since we observed that the active period of host initiating SQLi is short, it is important to constantly update the list from various sources.

In this report, we discuss some of the most popular tools as we outline the challenge of – and solutions to – SQLi attacks targeting Web applications.

[1] According to Privacyrights.org, from 2005 to today there were 312,437,487 data records lost due to hacking. About 262M records from TJMax, RockYou and Heartland, all SQL injection attacks.

SQL Injection

In a SQL Injection attack, attackers exploit a Web application vulnerability in order to access the organizations' data in an unauthorized manner. For laypeople, this means typing computer code in the fields of a website's form. For example, instead of typing in a credit card number or a last name, a hacker types in something technical that looks like 'x'='x'. When clever code is used, this action tricks the website into coughing up sensitive data.

In geek speak, SQL injection is a technique used to take advantage of non-validated input vulnerabilities to pass SQL commands through a Web application for execution by a backend database. Attackers take advantage of the fact that programmers often chain together SQL commands with user-provided parameters, and can therefore embed SQL commands inside these parameters. The result is that the attacker can execute arbitrary SQL queries and/or commands on the backend database server through the Web application.

Whichever explanation you prefer, the potential results can be disastrous. For example, attackers may be able to retrieve the organization's intellectual property, customer account information and other sensitive data. A successful SQLi attack may also allow the attacker to steal the site's administrator password, giving the attacker full control over the Web application. Other times, a compromised site can host attacker code which may lead site visitors to download malware (aka "Drive-by-Downloads"[3]). SQLi attacks also allow the manipulation of data, enabling – for example – the defacement of the Website.

A SQLi attack usually starts with identifying weaknesses in the applications where unchecked users' input is transformed into database queries. Goal-oriented attacks continue with abusing these weaknesses in a repetitious trial and error process in order to discover the exact structure of the application's database. The aim is to discover what sensitive and valuable information is stored in the database and how to extract it. In practice, this tedious process is usually automated and often based on widely known tools that let an attacker quickly and effortlessly identify and exploit applications' vulnerabilities.

Detailed Analysis

Attack Occurrences

We have monitored SQL Injection attack attempts over the last nine months against a set of 30 real web applications, of all sizes, across different industries. On average, we have identified 53 SQLi attacks per hour and 1,093 attacks per day. Since July we have observed a slight increase in SQLi attacks, averaging 71 attacks per hour and 1,589 per day.

		Average	Min	Max	Median	Standard Deviation
Attacks / hour	Since December 2010	53	1	7950	9	197
	Since July	71	1	4937	8	259
Attacks / day	Since December 2010	1093	44	21724	600	1909
	Since July	1589	106	8204	1162	1508

Table 1: Statistics of SQLi occurrences

The large standard deviation indicates significant fluctuations in the number of attacks. For example, we witnessed on most days applications suffered less than 3,000 attacks and occasionally less than 500. However, there were a few days where about 8,000 SQLi attempts were concentrated against the applications.

Analyzing the number of hourly and daily SQLi attacks in the last nine months, we can see a trend of 50-100 attacks per hour (as shown in Figure 1) and about 1,100 daily attacks on average, with an occasional spike. The concentration of a huge number of attack attempts during a short period of time is a clear indication that these attacks are automated.

[3] http://www.imperva.com/resources/glossary/drive-by-downloads.html

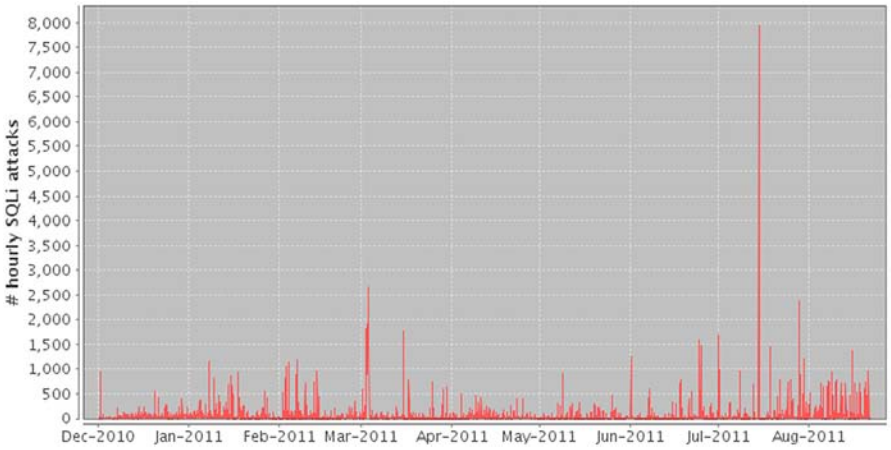

Figure 1: Hourly occurrences of SQLi attacks

Attack Tools

As mentioned, the high attack frequency suggests substantial usage of automated attack tools. We were able to identify the usage of a number of such tools based either on their HTTP User-Agent header or on well known characteristics of the injection vectors that these tools generate. Our data showed that each attacking host uses a single specific tool. Here are some of the tools we observed:

Sqlmap

Sqlmap[4] is an open source tool that automates the process of detecting and exploiting SQL injection flaws and taking over of database servers. It has many features including database fingerprinting, data fetching from the database, accessing the underlying file system, and executing commands on the operating system via out-of-band connections. The tool supports five SQL injection techniques: boolean-based blind, time-based blind, error-based, UNION query and stacked queries.

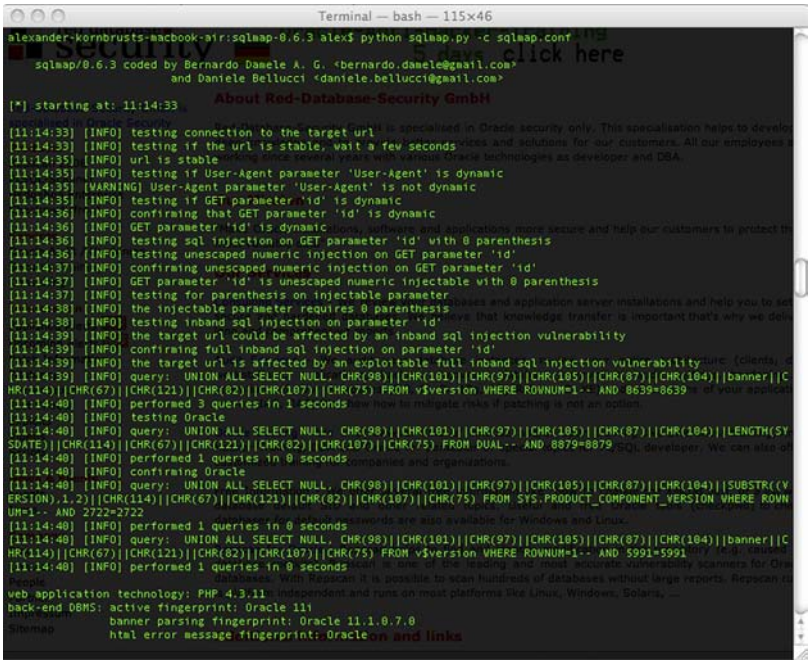

Figure 2: A screenshot of the Sqlmap attack tool

[4] See: http://sqlmap.sourceforge.net/

Havij

Havij[5] is a simple Windows GUI tool to automate SQL injection attacks. Its capabilities are similar to tools like *sqlmap* but it is more user-friendly. *Havij* is distributed by *itsecteam*, an Iranian security company. *Havij* injects a "UNION ALL SELECT" statement and keeps adding additional fields to the union query to work out how many columns are required. Each statement selects static "random" hex strings (which contains a distinguishable sequence of characters) to make it easy to identify them in the response.

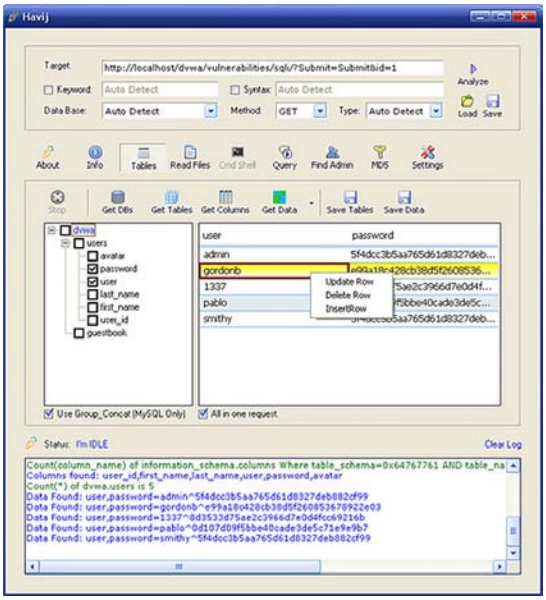

Figure 3: Screenshots of the Havij attack tool

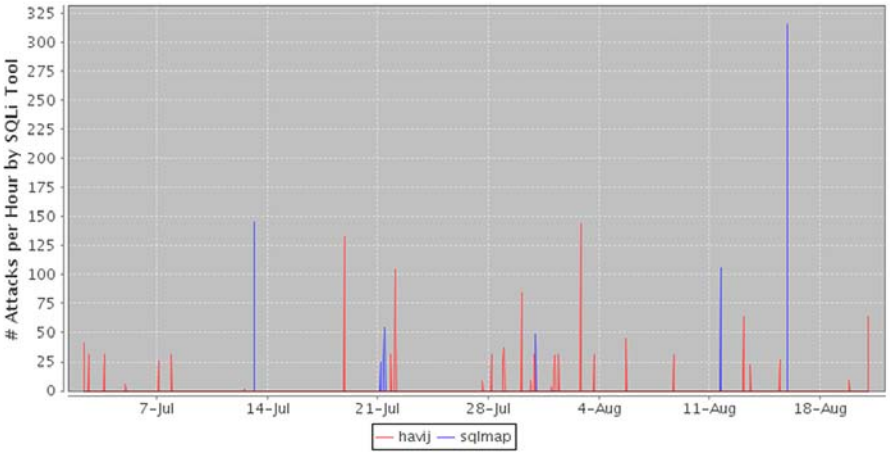

Figure 4: Hourly attacks using the Sqlmap and Havij tools since July

[5] See: http://itsecteam.com/en/projects/project1.htm

Attacking Hosts

The SQLi attacks we observed since July originated from 3,845 hosts. However, the distribution of activity between them is uneven: The top three hosts accounted for 23% of the attacks, and the next seven sources accounted for 18% of the attacks. However, five of these seven hosts are Akamai proxies, so the traffic was just routed through them from the attacker-controlled hosts.

During this observed period, each host generated an average of 12 attacks. However, five hosts generated 1000-2000 attacks over 1-7 consecutive days of activity, and 30 additional hosts generated over 100 attacks over 1-2 days of activity. This pattern suggests that creating and using a black list of hosts that initiated SQLi attacks (for example, through a community for sharing security information) may increase the ability to quickly identify and block such attackers. The average time each host was active during this period was half a day, and the median of the attack period was much lower. Therefore, the update rate of the black list must be high in order to keep up with new threats.

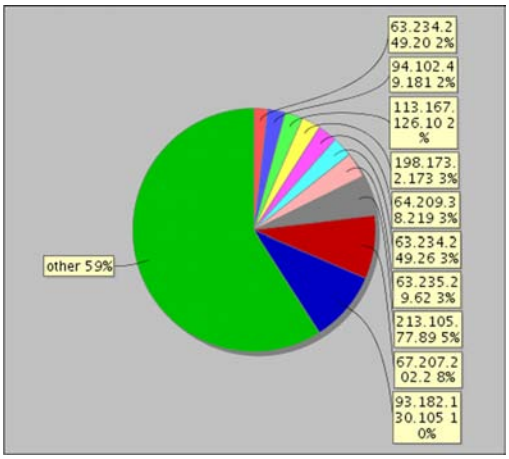

Figure 5: Highest-activity attackers since July

To corroborate our assumption that a significant number of SQLi attack attempts are made by automatic tools, we focused on eight hosts that have issued large numbers of attacks during ten minute time-frames since the beginning of August. During these periods, these hosts issue ten or more attack attempts per minute. Obviously, this is beyond the rate that can be done by a human operator. Furthermore, we have noticed that most of these hosts participated in several SQLi campaigns during this period, and that their malicious activity was limited to only SQLi attacks. Some of these hosts are servers that were compromised and used as a platform to launch the attacks.

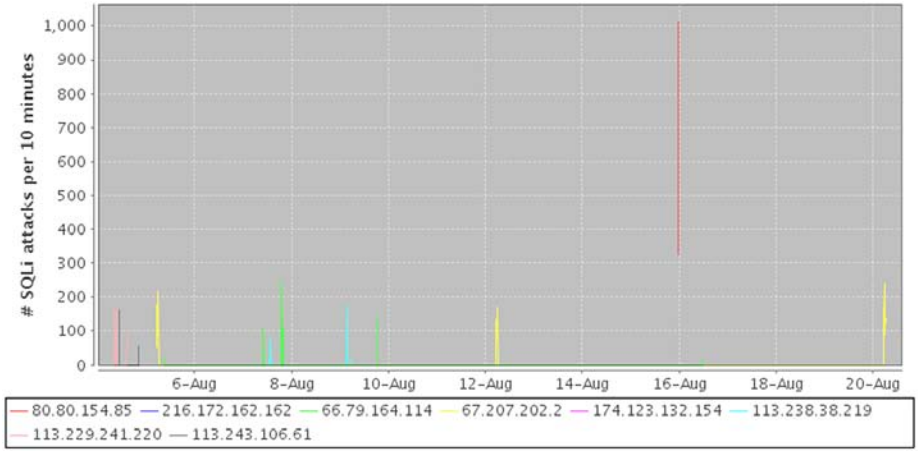

Figure 6: Automated attacks occurrences during August

Geographic Location of Attack Sources

Attackers use botnets, or compromised hosts, to execute planned attacks. As such, the geographic location of an attacking host does not necessarily point to the actual location of the attacker controlling the host and directing the attack. However, from a global security viewpoint, it may indicate that there is a large concentration of compromised hosts in a particular country that belong to botnets. It also means that a large deviation from the typical amount of traffic generated from specific countries may demonstrate that an attack is underway.

The SQLi attacks we observed originated from all around the world. However, most of the attacks originated from the United States. Hosts located in Sweden, China, Great Britain and Vietnam also heavily participated in SQLi attacks. (Table 2) Looking at the number of hosts from each country that initiated SQLi attacks[6], again we see that the United States leads, followed by China, Great Britain and Germany.

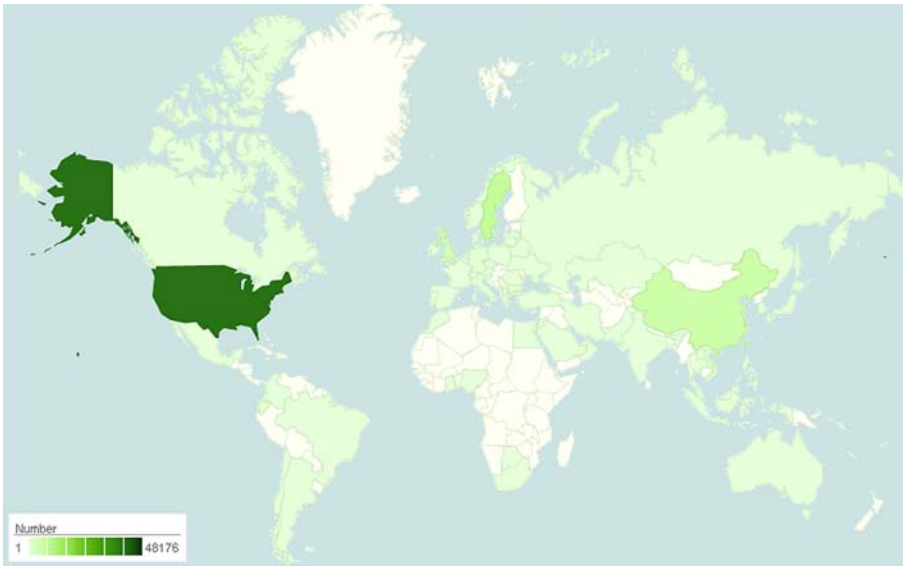

Figure 7: Countries of attacks' origins since July

Country	Number of Originating Attacks	%
United States	48176	58
Sweden	8850	11
China	6709	8
Great Britain	4970	6
Vietnam	2412	3
Netherlands	1963	2
Bulgaria	1359	2
Ecuador	1356	2
European Union	1093	1
Germany	971	1
Other	4748	6

Table 2: Countries of attacks' origins since July

[6] Note that the number of hosts that initiated attacks per country is not normalized, for example by the total number of hosts in this country.

Attack Vectors

We categorized the different attack vectors observed in the traffic based on the SQL capabilities they employ. This section explores some samples of these different exploitation mechanisms.

Direct Query Manipulation

A simple way for an attacker to modify the application's behavior is to bypass any filter on the results of an SQL query generated from the user's input: The attacker replaces the application-generated filter with a filter that always evaluate to Boolean *true* (or *false*, depending on the meaning of the query). This is often done by appending a logical expression with a known value to the parameter that the application expects, like: **`' OR 1=1`** to get a *true* value or **`1'/**/aND/**/'8'='3`** to get a *false* value. This type of vectors is most often used to establish the existence of a SQL injection vulnerability.

Discovering the Database Structure

More sophisticated attacks on the application's database depend on knowing the specific structure of the tables it uses. Assuming that the application exposes SQL errors it encounters, one of the ways to discover the number of columns in a table is using the SQL *Order By* clause: the database returns an error when the order parameter is an invalid column. The traffic shows that some attackers begin by sending: **`6' Order By 10000 /*`**. Since 10000 is an invalid identifier for a column, the attacker hopes to receive an error. Upon error, the attacker proceeds to decrease the parameter (e.g. **`6' Order By 50 /*`**) until the commands complete without an error (this is usually done in a binary search fashion to achieve improved efficiency). This, in turn, confirms that the current parameter for the ordering of the results is in fact the number of columns in the table.[7]

The process continues by querying the database metadata tables to discover the names of those tables and columns that may contain valuable information. An example of such observed attack contains the following input: **`2755') and 1=convert(int,(select top 1 table_name from information_schema.tables))--`**

Union Select SQL Injection

We have witnessed another popular attack vector which accounted for 9% of the SQLi attempts since July. This particular attack incorporates *Union Select* SQL statements. The UNION SELECT statement allows the chaining and retrieval of the results of two separate SQL SELECT queries that have nothing in common. Upon success, the injection of Union Select allows the attacker to retrieve data from a particular database table even though the application was designed to access another table.

Some of the observed attacks simply performed checks to see whether the application is vulnerable to Union Select injection, for example by injecting into an HTTP parameter the value: **`34 UNION SELECT 12345`**. In a more dangerous example, the following value was injected into an application's forum page in order to retrieve the personal details of a user, for which the attacker guessed the application-given identifier: **`XXXX%') UNION SELECT MEMBER_ID, M_STATUS, M_NAME + '/' + M_EMAIL + '/', M_LEVEL, M_EMAIL, M_COUNTRY, M_HOMEPAGE, M_ICQ, M_YAHOO, M_AIM, M_TITLE, M_POSTS, M_LASTPOSTDATE, M_LASTHEREDATE, M_DATE, M_STATE FROM FORUM_MEMBERS WHERE (M_NAME LIKE '`**. Note that this is a very specific injection. An attacker can perform this type of attack only after discovering the detailed structure of the application's database tables. Alternatively, the attacker can perform this attack by assuming that the deployed application is a popular one, and basing the attack on its previously known database schema. In this example, the attack is based on the published database structure of Snitz Forums 2000, an open source bulletin board system[8].

[7] See a detailed description of the process in: http://worldwidehack.com/viewtopic.php?f=26&t=7
[8] See: http://forum.snitz.com/specs.asp

Time-based blind SQL injection

The *time-based blind SQLi*[9] technique bypasses some security measures that Web administrators place on their systems. It is a popular variant of SQL Injection and we witnessed it in 13% of the SQLi attempts since July. Sometimes, when an attacker executes a SQL Injection attacks, the server responds with error messages from the database server complaining that the SQL query syntax is incorrect. To prevent exposing any information about the database to an attacker, secure applications respond with a generic prepared error page. This makes exploiting a potential SQL Injection vulnerability more difficult but not impossible. *Blind SQL injection* is a technique for stealing data from the database by asking a series of True and False questions through SQL statements. An attacker may verify whether a sent request returned *True* or *False* in a few ways. One of these methods is using a time consuming operation, e.g. "*waitfor delay*", that will delay the server responses if the expression is *True*.

For example, we have observed injection into a web form's returned HTTP parameter of the value: **`x' wAiTfOr dELay '0:0:20'--`**. In this case, the attacker checks if the application is vulnerable to SQLi. If the application and its database accept the injected value x, it will also wait for 20 seconds – a noticeable delay that the attacker's tool can identify, marking this parameter as vulnerable to injection. In a similar but more specific usage, we have observed injection into the New Customer parameter of a Login form of the value: **`No');waitfor delay '0:0:05';--`**. In this case the attacker attempts to bypass the application's login flow, with the hopes that a successful injection (identified by the five seconds delay he specified) will let him impersonate an existing user.

As a side note, injection of *waitfor delay* can be used by an attacker to tie up resources in the application and cause Denial of Service.

Bypassing simple parameter sanitation

Attackers are aware that applications usually sanitize the user's input before creating a database query from it. However, if this validation of the input is naive, an attacker can bypass it by simple means like inserting SQL-ignored comments (denoted by /* and */) into the input string (e.g. **`1'/**/aND/**/'8'='3`**), switching between lower-case and upper-case characters (e.g. **`x' wAiTfOr dELay '0:0:20'--`**), or using SQL string functions like concat() and char() to create the value for injection indirectly, without triggering the application's injection detector. Another frequent evasion technique in the traffic is to encode the injected command using its characters' ASCII values, like: **`1 DeClARe @x varchar(99) set @x=0x77616974666f722064656c61792027303a303a323027 exec(@x)--`** which the database translates and understands as: **`1 waitfor delay '0:0:20'--`**.

[9] http://www.imperva.com/application_defense_center/white_papaers/blind_sql_server_injection.html, https://www.owasp.org/index.php/Blind_SQL_Injection

Recommendations

SQL Injection is one of the most prevalent attack types today. With SQL-Injection tool kits publically available, or traded in underground markets, even a non-proficient attacker can carry out such an attack. Recent widely publicized "success stories" of SQL Injection attacks will probably increase attackers' efforts in this direction.

A successful SQLi attack allows the hacker to penetrate the target application's defenses and gain access to its sensitive information. Companies and organizations that suffered huge monetary losses and public embarrassment are no longer a rarity, but instead are an unpleasant example of what happens when attackers' determination combine with insufficient security measures. What can organizations do to prevent SQL Injection attacks?

Application code can be written in a manner that defeats SQL injection. However, ensuring that each and every piece of database access code is immune to SQL injection in normal size applications that also include 3rd party components is essentially impossible. One must therefore assume that deployed web applications probably do include SQL injection vulnerabilities. Given the nature of attack tools that explore all potential parts of the application, it seems that sooner or later these vulnerabilities will be detected and exploited unless a run-time SQL injection detection mechanisms exits.

Signature based detection mechanisms have often failed to detect SQL injection due to the constant evolution of attack techniques as well as evasion techniques. Therefore, we recommend a three folded approach to defeat SQL injection attacks in real time:

- **Detect SQL injection attack.** Using a combination of application layer knowledge (application profile) and a preconfigured database of attack vector formats. Detecting SQLi must normalize the inspected input to avoid evasion attempts.
- **Identify access patterns of automated tools**. In practice, SQLi attacks are mostly executed using automatic tools. Various mechanisms exist to detect usage of automatic clients, like rate-based policies and enforcement of valid client response to challenges.
- **Create and deploy a blacklist of hosts that initiated SQLi attacks**. This measure increases the ability to quickly identify and block attackers. Since we observed that the active period of host initiating SQLi is short, it is important to constantly update the list from various sources.

To reduce the potential effect of a successful SQLi attack, the content of the database should be analyzed to verify that the application only has access to the minimal data it actually needs. This should minimize the exposure of the organization in case of a penetration to the database through the application.

Hacker Intelligence Initiative Overview

The Imperva Hacker Intelligence Initiative goes inside the cyber-underground and provides analysis of the trending hacking techniques and interesting attack campaigns from the past month. A part of Imperva's Application Defense Center research arm, the Hacker Intelligence Initiative (HII), is focused on tracking the latest trends in attacks, Web application security and cyber-crime business models with the goal of improving security controls and risk management processes.

Imperva
3400 Bridge Parkway, Suite 200
Redwood City, CA 94065

Tel: +1-650-345-9000
Fax: +1-650-345-9004
www.imperva.com

© Copyright 2011, Imperva
All rights reserved. Imperva, SecureSphere, and "Protecting the Data That Drives Business" are registered trademarks of Imperva.
All other brand or product names are trademarks or registered trademarks of their respective holders. #HII-SEPTEMBER-2011-0911rev1