An Analytical Scanning Technique to Detect and Prevent the Transformed SQL Injection and XSS Attacks

Mohammad Qbea'h[1], Saed Alrabaee[1] and Djedjiga Mouheb[2]
[1]*Information Systems and Security, College of IT, United Arab Emirates University, Al Ain, Abu Dhabi, U.A.E.*
[2]*Department of Computer Science, College of Computing and Informatics, University of Sharjah, U.A.E.*

Keywords: SQL Injection Attack, Cross Site Scripting, Encoding, Base64 Encoding, XSS, Hex Encodings, SQLIA, Android Attack.

Abstract: Among the most critical and dangerous attacks is the one that exploits Base64 or Hex encoding technique in SQL Injection (SQLIA) and Cross Site Scripting (XSS) attacks, instead of using plain text. This technique is widely used in most dangerous attacks because it evades detection. Therefore, it is possible to bypass many filters such as IDS, without taking into account the transformation methodologies of the symbols and characters. Moreover, it reserves the same semantics with different syntax. Attackers can exploit this serious technique to reach unseen data and gain valuable benefits. To the best of our knowledge, this paper presents the first technique that focuses on detecting and preventing transformed SQLIA and XSS from Base64 and Hex encoding. We perform scanning and analyzing methods by targeting two places: (i) Input boxes and (ii) Strings in page URLs. Then, we decode the inputs and compare them with our stored suspicious tokens. Finally, we perform string matching and mutation mechanisms to revoke the activity of malicious inputs. We have evaluated our technique and the results showed that it is capable to detect and prevent this transformed attack.

1 INTRODUCTION

We have experienced a noticeable growth of SQL injection attacks against web applications over the last two years. Such serious attacks should worry webmasters and website owners (ZAkamai, 2019). Millions of websites and systems in the world use backend servers to store billions of records of important data in relational databases. This data could be very critical and sensitive such as military, medical, financial and personal data. Unfortunately, the importance of this data attracts attackers from everywhere to spare extreme efforts to break the bridges in order to access this hidden data and gain valuable benefits. Furthermore, the attackers can steal a lot of money by accessing the entire database and getting the credit card's numbers.

The increasing attempts to destroy and damage the structure and the content of databases should be given very high attention. This is because attackers use very dangerous attacking techniques, which are SQL Injection Attack (SQLIA) and Cross-Site Scripting (XSS) to target sensitive institutes (Balasundaram and Ramaraj, 2012; Khoury et al., 2011; Patel and Shekokar, 2015; Qbea'h et al., 2016; Sadeghian et al., 2013; Wu et al., 2011).

One of the critical consequences of a successful attack is affecting the integrity of the database by tampering data records to become inaccurate and inconsistent (OWASP, 2016). This could lead to sending and receiving wrong information and transactions to wrong targets. In addition, the weaknesses of input sanitization are the most significant causes of SQLIA (Sathyanarayan et al., 2014). Moreover, we find that there are a lot of free tools, mainly open source, such as, Drupal, Joomla, WordPress and Quest that have vulnerable components, which are a source of SQLIA and XSS.

There are many specialized institutes in software security that publish a lot of reports to address common and critical vulnerabilities in the web, software and mobile applications. Some of these reports can classify and order the vulnerabilities based on their risk. These vulnerabilities can be exploited by attackers to steal important information or to damage the back-end server or even to break the authentication to the database. We classify these important studies and reports as follows:

Qbea'h, M., Alrabaee, S. and Mouheb, D.
An Analytical Scanning Technique to Detect and Prevent the Transformed SQL Injection and XSS Attacks.
DOI: 10.5220/0009004006030610
In *Proceedings of the 6th International Conference on Information Systems Security and Privacy (ICISSP 2020)*, pages 603-610
ISBN: 978-989-758-399-5; ISSN: 2184-4356

1.1 Critical Vulnerabilities in the Web, Software Application

1.1.1 Web Attacks and Gaming Abuse

A recent study by Akamai Company in the last two years (between November 2017 and March 2019) has published a report named ”Web Attacks and Gaming Abuse (Volume 5, Issue 3) Akamai”, which shows that SQL injection in these days represents two-thirds (65.1%) of all web application attacks. The report has also included a country ranking study based on the top ten targets and top ten sources of application attacks. As a result, the United States ranked first among the ten surveyed countries as a target of 2,666,156,401 attacks and as a source of 967,577,579 attacks (ZAkamai, 2019; Darkreading, 2019).

1.1.2 Web Application Vulnerability

Acunetix, a pioneer in the field of automated web application security and leader in technology, has published a web application vulnerability report in 2019. It has analyzed the vulnerabilities detected in year 2019, across 10,000 scan targets and one of the results is that 30% of web applications are vulnerable to XSS (Acunetix, 2019). In addition, White-Hat Security Threat Research Center (TRC) has validated and analyzed millions of several attacks in multiple market sectors. The report classified cross-site scripting attack as one of the top 10 vulnerabilities (WhiteHat, 2019).

1.1.3 A State of Software Security

Veracode company has published a state of software security report in 2018. It stated that attackers can use any vulnerability for SQLI to gain unauthorized access to a database server by using maliciously crafted input, so they have high exploitability ratings. Also, many programming languages such as JAVA, DOT NET and C++ are still rife with vulnerable components. Moreover, one of the three applications are injected against SQL injection flaws. On the other side, 50% of applications have XSS vulnerabilities (Veracode, 2019).

In this paper, we present a first technique that focuses on detecting and preventing the transformed forms of SQLI and XSS together from Base64 and Hex encoding. Our technique depends on performing several steps: (i) Scanning: we scan two targets of inputs, namely input box and page URL. Then, we split the input into tokens. (ii) Decoding: we decode the transformed inputs from base64 or hex encoding. After that, we search about any suspicious tokens. (iii) Analyzing. (iv) Matching. (v) Replacing. (vi) Triggering excepting handling. Finally, we get the final input format with free malicious attack.

Most of existing works focused on detecting or preventing SQLI or XSS attacks without taking into consideration the other shapes of these attacks such as Base64 and Hex encoding. In addition, to the best of our knowledge, the encoded XSS attack has not been considered in many of the surveyed research papers.

The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 presents our proposed technique. In Section 4, we analyze transformed SQLI and XSS attacks. Section 5 and Section 6 present the experiments and implementation. The technical details of the proposed technique are discussed in Section 5.2. Finally, we conclude in Section 7.

2 RELATED WORK

In this section, we summarize a number of techniques, methods and approaches, which were used to address SQLIA and XSS. To the best of our knowledge, only a few number of papers have addressed the transformation of SQLI and XSS attacks using encoding techniques. Also, there were no rich discussions about how to prevent such encrypted attacks, especially XSS.

SQLRand (Boyd and Keromytis, 2004) proposed a randomization technique used to encrypt the keywords of SQL to detect SQLIA. However, this technique couldn’t detect and prevent the transformed SQLIA, and it needs to remember the keyword. In, the authors proposed a method to prevent SQL injection by splitting the input query into tokens based on single quote, space, and double dashes. After that, a comparison is made to accept or reject the query. However, the method did not take into account the transformation techniques into ASCII or Unicode. In (Wassermann and Su, 2004), the authors proposed a technique using an automated reasoning with a static analysis. The technique is efficient but it handled only tautology SQLIA, and it couldn’t detect or prevent the other types of SQLIA or XSS. The authors in (Prabakar et al., 2013) proposed a technique based on static and dynamic phases using pattern matching algorithm to prevent SQL injection. This technique is effective, however, it addressed some samples of standard attacks but not all SQLIA. Also, the technique did not discuss XSS and the encoded attacks. In (Balasundaram and Ramaraj, 2011), the authors proposed a technique that uses an authentication scheme to prevent SQLIA based on both Advanced Encryption Standard (AES) and Rivest-Shamir-Adleman (RSA). However, this technique requires to maintain every user secret key at the server side and the client side. Moreover, it does not support URL-based SQLIA or XSS. In (Qbea'h et al., 2016), the authors proposed an effective formal approach based on finite automata with regular expression to detect and prevent tautology and alternate encoding of SQLIA. However, their approach didn't address XSS neither the transformed SQLIA using Base64 encoding technique.

The authors in (Buehrer et al., 2005) proposed SqlGurd, a technique based on parse tree comparison. This technique is limited to only plain text of SQLIA without any discussion about transformed SQLIA. In addition, XSS was not considered at all. In (Bisht et al., 2010), the authors proposed CANDID technique, which uses a candidate inputs to detect SQLI using a comparison between the structures of query of candidate input and the potential SQLIA. However, the technique has a number of limitations and can fail in some cases, such as: First, An exception can be triggered when a function is considered as a non-candidate especially if it contains an 'a' character. Second, if a white space appeared without classifying it in the right implementation level, then exceptions can happen and could make the technique fail. Third, there is a difficulty to decide if the keywords are intended from the programmer or not. Furthermore, this technique focused on a normal form of SQLI rather than the transformed one. Also, there is no discussion regarding XSS. In (Junjin, 2009), the authors proposed an automated approach based on static analysis to detect SQLI vulnerabilities and finding bugs. However, this approach did not take into consideration SQLIA prevention. In addition, transformed SQLIA and XSS were not addressed. Other efforts have been conducted in discovering such vulnerabilities in binary code (Alrabaee et al., 2018; Alrabaee et al., 2015).

3 PROPOSED TECHNIQUE

In this section, we present a technique for scanning and analyzing the inputs by focusing on two sources: input boxes and query strings in page URLs. We apply our technique on the following targets: websites, applications and mobile apps. Then, we decode the input from Base64 or Hex into its origin (plain text). After that, we compare the decoded input with our stored suspicious tokens to detect the attack. Finally, we perform string matching and replacing mechanism in order to prevent the transformed SQLIA and XSS from Base64 and Hex encoding. The technical details of the proposed technique are discussed in Section 5.2.

Table 1 and Table 2 list samples of the most potential SQLI commands and XSS codes that can be applied by attackers through two sources: input boxes and page URLs'.

Table 1: Tautology SQLIA strings payload samples.

999'or 7=7--
manager' or '9'='9'--
manager' or 'z'='z'#

Table 2: XSS strings payload samples.

$< script > alert('attack'); < /script >$
$< script > alert(\backslash XSSAttack"); < /script >$
$< script > alert(111) < /script >$

4 ANALYZING TRANSFORMED SQLI AND XSS

4.1 Base64 Encoding Steps

The following are the steps for Base64 encoding:

1. Splitting the binary code from 8 bits into 6 bits
2. Convert the 6 bits into decimal digits
3. Map the obtained decimal number with the following series (starting from 0 to 63):

```
ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghi
jklmnopqrstuvwxyz0123456789+/
```

4.2 Analyzing Base64 Encoding Technique with XSS

We use the script $< script > alert('attack');$ $< /script >$ in lower case formatting to perform our demonstration, but the attacker can transform the XSS to any case sensitivity format. Here are some examples: $< SCriPT > ALerT('attack'); < /script >$ $, < scRIPT > ALERT('attack'); < /SCRIPT >$, etc. Table 3 shows that the attacker can transform the injected command into something unknown by using Base64 encoding technique. Therefore, we generate the following decoded phrase:

```
PHNjcmlwdD5hbGVydCgnYXR0YWNrJyk7PC9
zY3JpcHQ+
```

instead of the plain text format:
" $< script > alert('attack'); < /script >$ ".

Table 3: Analyzing XSS using Base64 encoding technique.

Input	<script>alert('attack');</script>								
Splitting	<	s	c	r	i	p	t	>	..
Decimal	60	115	114	99	105	112	116	62	..
Hex	3C	73	72	63	69	70	74	E3	..
Binary (8-bits)	00111100	01110011	01110010	01100011	01101001	01110000	00110101	00111110	..
Full binary code	001111000111001101100011011100 100110100101110000011101000011 111001100001011011000110010101 110010011101000010100000100111 011000010111010001110100011000 010110001101101011001001110010 100100111011001111000010111101 110011011000110111001001101001 0111000001110100001111110								
Decimal	157133528383748293573327621 502923239242317522422134395 036591526151245594128716662								
Encoding	PHNjcmlwdD5hbGVyd CgnYX R0YWNrJyk7PC9zY3JpcHQ+								

4.3 Analyzing Hex Encoding Technique with SQLIA

We analyze this command 'or $'@' =' @' - -$ of SQLIA as Hex Encoding in a page URL. URL encoding or percent encoding method can be applied by substituting characters with a percent notation (%) followed by two ASCII values (Paige, 2013). Table 4 shows, in details, the analysis of transformed SQLIA using Hex encoding. As a result, the transformed SQLIA from Hex encoding in URL is: %27or%20%27@%27%3D%27@%27−−

Table 5 shows a comparison between our technique and the available techniques in terms of detecting and preventing transformed SQLIA and XSS using Base64 and Hex encoding. As a result, our proposed technique is able to detect and prevent the transformed SQLIA and XSS while other techniques either did not address that or failed to control such attacks.

5 EXPERIMENTS

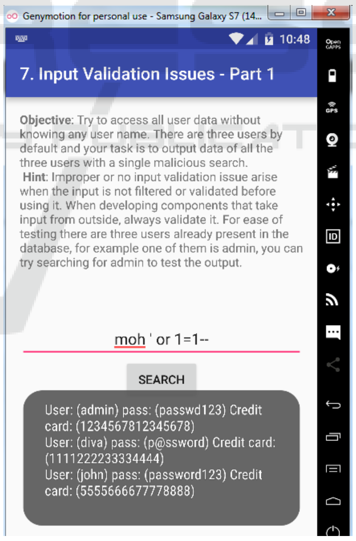
In this section, we present how to detect and prevent the transformed SQLIA and XSS from Base64 and Hex encoding. We apply our experiment on the several testing environments such as: SQL Injection Ninja Testing Lab (SecurityIdiots, 2012), XSS-Game (XSS, 2019), Damn Vulnerable Web App (DVWA) (DVWA, 2015), Damn insecure and vulnerable App (DIVA) (DIVA, 2016), and our local testing application. We select two applications as testing environments.

5.1 Attack Detection

5.1.1 Damn Insecure and Vulnerable App (DIVA)

Damn insecure and vulnerable App (DIVA) is an android application developed to teach security professionals and programmers to protect applications and handle insecure or vulnerable code. This simulation works under Android operating system and can cover several vulnerabilities such as SQLIA.

Figure 1 shows a successful attack using SQLIA on DIVA Android application using this malicious code: *moh*$'$ or $1 = 1 - -$. We execute our injected code to gain access to all users with their credit card numbers.



7. Input Validation Issues - Part 1

Objective: Try to access all user data without knowing any user name. There are three users by default and your task is to output data of all the three users with a single malicious search.
Hint: Improper or no input validation issue arise when the input is not filtered or validated before using it. When developing components that take input from outside, always validate it. For ease of testing there are three users already present in the database, for example one of them is admin, you can try searching for admin to test the output.

moh ' or 1=1--

SEARCH

User: (admin) pass: (passwd123) Credit card: (1234567812345678)
User: (diva) pass: (p@ssword) Credit card: (1111222233334444)
User: (john) pass: (password123) Credit card: (5555666677778888)

Figure 1: Successful attack using SQLIA on Android Application.

5.1.2 Damn Vulnerable Web App (DVWA)

DVWA is a free open source environment that provides penetration testing methods for learning purposes and to simulate web vulnerabilities such as XSS and SQLIA. It uses PHP and MySQL.

Figure 2 shows a successful attack using transformed XSS on DVWA application using this mali-

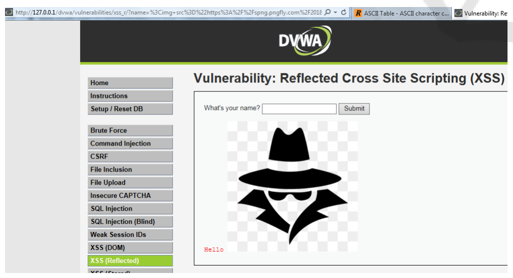
Table 4: Analyzing SQLIA using Hex encoding technique.

Input	‘	o	r	SPACE	‘	@	‘	=	‘	@	‘	-	-
URL Hex Encoding	27%	o	r	20%	27%	@	27%	%3D	27%	@	27%	-	-

Table 5: A comparative capable to detect and prevent the transformed SQLI and transformed XSS Attacks from Base64 and Hex encoding.

Approach/Technique	Transformed SQLI Detection	Transformed SQLI Prevention	Transformed XSS Detection	Transformed XSS Prevention	Rich Examples
SQLRAND	NO	NO	NO	NO	NO
SQL GUARD	Partial	NO	NO	NO	NO
CANDID	NO	NO	NO	NO	NO
SQL CHECK	Partial	NO	NO	NO	NO
DIWEDA	NO	NO	NO	NO	NO
SQLIPA	NO	NO	NO	NO	NO
Automated Approach	NO	NO	NO	NO	NO
Proposed Technique	Yes	Yes	Yes	Yes	Yes

cious code: %3Cimg+src%3D%22https%3A%2F% .jpg%22%2F%3E Then, we execute our injected code, which has critical transformed symbols such as %2F instead of using plain text of slash (/). As a result, a malicious image is inserted and appeared on the webpage, thus the website is attacked using transformed XSS.


http://127.0.0.1/dvwa/vulnerabilities/xss_r/?name=%3Cimg+src%3D%22https%3A%2F%2Fspng.pngfly.com%2F2018

What's your name?

Hello

Figure 2: Successful attack using Transformed XSS using image tag.

5.2 Attack Prevention

In this section, we perform our technique for preventing the transformed XSS and SQLIA. We perform our technique through six steps: i) Scanning; ii) Decoding; iii) Analyzing; iv) Matching; v) Replacing or Mutating; vi) Triggering Exception handling. We start explaining our work as follows: First, we perform a full scanning on the query string in either user input or page URL by splitting the encoded symbols into tokens. Then, we apply decoding technique on these tokens by transforming them from base64 or hex format into their original format. After that, we search about any suspicious tokens. Then, we analyze these tokens by classifying and marking them as either suspicious (S) or normal (N). We achieve that by applying a matching mechanism between the transformed symbols and characters with our stored suspicious symbols and characters. Next step, we mark the matched symbols as (M) and (NM) is marked for Not Matched symbols. Finally, we perform replacing and mutating mechanisms by substituting any symbol or character which is marked as suspicious (S) with an empty or NULL value. We add one extra layer to handle any exception that may appear by triggering a caution message to inform the attacker that he is under monitoring. As a result, we get a string format with free suspicious tokens and without any malicious activity.

By applying our technique, we not only freeze the attack but also inject the malicious code with a drug that can mutate the genetic of all transformed attacks. This is similar to an action that could happen in natural life when we revoke a snake venom without killing the snake. In addition, for further protection, an extra layer is added to our technique by generating a caution message using exception handling method. This caution should appear to attackers to inform them to stop immediately using the injection because they are under legal liability. We select a sample of transformed SQLIA:"*J29yIDU9NS0t*" to demonstrate our technique as shown in Table 6. The final transformation of the transformed SQLIA is "*orSPACE55*" and it is equal to "*or55*". Based on the final transformation, we can consider this result as normal and safe format.

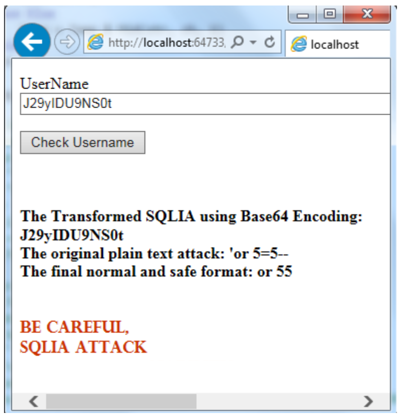
6 IMPLEMENTAION

To implement and simulate our technique, we have used Microsoft Dot Net, VB.NET and Microsoft SQL Server as testing environments. We build a very strong code in VB.NET to prevent the Transformed SQLIA and XSS attacks from Base64 and Hex encoding. We provide our code to developers, security analysts and programming language designers to write a free vulnerable code against SQLIA and XSS. This code is general and can be programmed in any programming language such as: JAVA, C++, PHP and Python. In this section, we provide the code used for implementing our technique.

6.1 Preventing Transformed SQLIA from Base64 Encoding

In Figure 3, we used the malicious code "*J29yIDU9NS0t*" to test and simulate our technique. We have succeeded to prevent Transformed SQLIA from Base64 encoding. We provide our code that we used to detect and prevent transformed SQLIA from Base64 encoding as follows:

```
Dim base64E As String=txtName.Text
Dim base64D As String,
Dim sentence() As Byte
sentence = System.Convert.
FromBase64String(base64E)
base64D=System.Text.ASCIIEncoding.
ASCII.GetString(sentence)
Dim result1 = base64D
Dim strBase64() As String={"′", "=",
 "-", ">","<",";", "(", ")", "/"}
For Each i as char In strBase64
result1=result1.ToString.
Replace(i, "").ToLower().Trim()
Next
ResultLabel.Text =
"<br/>"+"<br/>"+"The Transformed
SQLIA using Base64 Encoding: "
+txtName.Text+"<br/>"+"The original
plain text attack:"+ base64D+"<br/>"
+"The final normal and safe format:"
+result1.ToString().ToLower.Trim()
ResultLabel0.Text="SQLIA Attack"
```


http://localhost:64733/ localhost

UserName

J29yIDU9NS0t

Check Username

The Transformed SQLIA using Base64 Encoding: J29yIDU9NS0t
The original plain text attack: 'or 5=5--
The final normal and safe format: or 55

BE CAREFUL, SQLIA ATTACK

Figure 3: Successful Prevention of Transformed SQLIA from Base64 Encoding.

6.2 Preventing Transformed XSS from Hex Encoding

In Figure 4, we used the malicious code

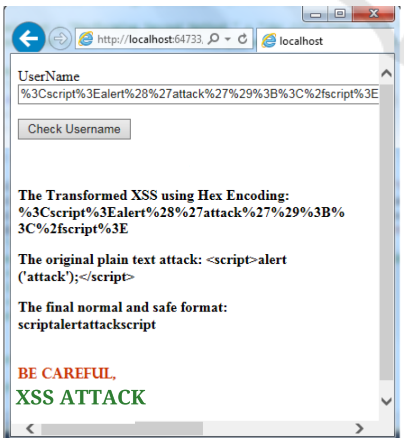
```
%3Cscript%3Ealert%28%27attack
%27%29%3B%3C%2fscript%3E
```

to test and simulate our technique. We succeeded to prevent the transformed XSS from Hex Encoding. We provide our code that we used to detect and prevent transformed XSS from Hex Encoding as follows:

```
Dim str() As String={"′", "=", "-",
">","<", ";", "(", ")", "/", "+"}
Dim str3
str3=HexDecoding(txtName.
Text.ToString.ToLower.Trim())
Dim result
result=HexDecoding(txtName.
Text.ToString.ToLower.Trim())
Dim i as char
For Each i In str
result = result.ToString.
Replace(i, "").ToLower().Trim()
Next
ResultLabel.Text="<br/>"+"<br/>"+
"The Transformed XSS using Hex
Encoding:"+"<br/>"+txtName.Text
ResultLabel1.Text="The original plain
text attack:"+
Server.HtmlEncode(str3)
ResultLabel2.Text="The final normal
and safe format:"+
"<br/>"+ result.
ToString().ToLower.Trim()
ResultLabel0.Text=" XSS Attack"
```

Table 6: Applying our technique to prevent transformed sqlia.

Scanning												
Symbols	J	2	9	Y	I	D	U	9	N	S	0	t
Decoding												
Decimal	9	54	61	50	8	3	20	61	13	18	52	45
Binary (6-bits)	0010 01	110 110	1111 01	1100 10	0010 00	00 00 11	01 01 00	11 11 01	00 11 01	01001 0	11010 0	101101
Binary (8-bits)	0010 0111	011 011 11	0111 0010	0010 0000	0011 0101	00 11 11 01	00 11 01 01	00 10 11 01	00 10 11 01	NULL	NULL	NULL
Hex	27	6F	72	20	35	3D	35	2D	2D	NULL	NULL	NULL
Decimal	39	111	114	32	53	61	53	45	45	NULL	NULL	NULL
Splitting characters	’	o	r	SPACE	5	=	5	-	-	NULL	NULL	NULL
Analyzing												
Symbols	‘	o	r	SPACE	5	=	5	-	-	NULL	NULL	NULL
	S	N	N	N	N	S	N	S	S	NULL	NULL	NULL
Matching												
Potential Suspicious Symbol	‘	‘	‘	‘	‘	‘	‘					
Flag	S	S	S	S	S	S	S					
Our stored Suspicious Symbols	=	>	/	#	-	!	‘					
Flag	S	S	S	S	S	S	S					
Matching	NM	NM	NM	NM	NM	NM	M					
Replacing												
Replacing	NULL	o	r	SPACE	5	NU LL	5	NU LL	NU LL			
	N	N	N	N	N	N	N	N	N			
Exception handling												
Be careful, you are trying to use SQLIA attack and this work is under legal liability												



UserName
%3Cscript%3Ealert%28%27attack%27%29%3B%3C%2fscript%3E

The Transformed XSS using Hex Encoding:
%3Cscript%3Ealert%28%27attack%27%29%3B%3C%2fscript%3E

The original plain text attack: <script>alert('attack');</script>

The final normal and safe format:
scriptalertattackscript

BE CAREFUL,
XSS ATTACK

Figure 4: Successful Prevention of Transformed XSS from Hex Encoding.

7 CONCLUSION

We have presented a technique for detecting and preventing the transformed SQLIA and XSS from Base64 and Hex encoding by performing several steps: scanning, decoding, splitting, analyzing, matching, mutating and replacing. We have tested our work on free online labs and using a local application. The experiment showed that our technique is comprehensive and effective compared with other techniques especially that we address the transformed SQLIA and XSS together. In fact, very few papers have addressed this critical attack according to a lot of recent studies and reports issued by specialized companies and institutes in computer security. Furthermore, we have provided our code for developers, security specialists and programming language designers. As a result, we have succeeded to generate a comprehensive and practical code to support our analytical technique to detect and prevent the transformed SQLIA and XSS and to perform real life simulations. Moreover, our code can be used as supplementary materials to facilitate future research.

REFERENCES

Acunetix (2019). Web Application Vulnerability Report. https://cdn2.hubspot.net/hubfs/4595665/Acunetix_web_application_vulnerability_report_2019.pdf.

Alrabaee, S., Shirani, P., Wang, L., and Debbabi, M. (2015). Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71.

Alrabaee, S., Shirani, P., Wang, L., and Debbabi, M. (2018). Fossil: a resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):8.

Balasundaram, I. and Ramaraj, E. (2011). An authentication scheme for preventing sql injection attack using hybrid encryption (psqlia-hbe). *European Journal of Scientific Research*, 53(3):359–368.

Balasundaram, I. and Ramaraj, E. (2012). An efficient technique for detection and prevention of sql injection attack using ascii based string matching. *Procedia Engineering*, 30:183–190.

Bisht, P., Madhusudan, P., and Venkatakrishnan, V. (2010). Candid: Dynamic candidate evaluations for automatic prevention of sql injection attacks. *ACM Transactions on Information and System Security (TISSEC)*, 13(2):14.

Boyd, S. W. and Keromytis, A. D. (2004). Sqlrand: Preventing sql injection attacks. In *International Conference on Applied Cryptography and Network Security*, pages 292–302. Springer.

Buehrer, G., Weide, B. W., and Sivilotti, P. A. (2005). Using parse tree validation to prevent sql injection attacks. In *Proceedings of the 5th international workshop on Software engineering and middleware*, pages 106–113. ACM.

Darkreading (2019). SQL Injection Attacks Represent Two-Third of All Web App Attacks. https://www.darkreading.com/attacks-breaches/sql-injection-attacks-represent-two-third-of-all-web-app-attacks/d/d-id/1334960.

DIVA (2016). Damn insecure and vulnerable App. https://github.com/payatu/diva-android.

DVWA (2015). Damn Vulnerable Web App (DVWA). http://www.dvwa.co.uk.

Junjin, M. (2009). An approach for sql injection vulnerability detection. In *2009 Sixth International Conference on Information Technology: New Generations*, pages 1411–1414. IEEE.

Khoury, N., Zavarsky, P., Lindskog, D., and Ruhl, R. (2011). Testing and assessing web vulnerability scanners for persistent sql injection attacks. In *proceedings of the first international workshop on security and privacy preserving in e-societies*, pages 12–18. ACM.

OWASP (2016). Web Application Vulnerability Report. https://www.owasp.org/index.php.

Paige, M. (2013). The tangled web: A guide to securing modern web applications by michal zalewski. *ACM SIGSOFT Software Engineering Notes*, 38(4):39–40.

Patel, N. and Shekokar, N. (2015). Implementation of pattern matching algorithm to defend sqlia. *Procedia Computer Science*, 45:453–459.

Prabakar, M. A., Karthikeyan, M., and Marimuthu, K. (2013). An efficient technique for preventing sql injection attack using pattern matching algorithm. In *2013 IEEE International Conference ON Emerging Trends in Computing, Communication and Nanotechnology (ICECCN)*, pages 503–506. IEEE.

Qbea'h, M., Alshraideh, M., and Sabri, K. E. (2016). Detecting and preventing sql injection attacks: a formal approach. In *2016 Cybersecurity and Cyberforensics Conference (CCC)*, pages 123–129. IEEE.

Sadeghian, A., Zamani, M., and Ibrahim, S. (2013). Sql injection is still alive: a study on sql injection signature evasion techniques. In *2013 International Conference on Informatics and Creative Multimedia*, pages 265–268. IEEE.

Sathyanarayan, S., Qi, D., Liang, Z., and Roychoudary, A. (2014). Sqlr: Grammar-guided validation of sql injection sanitizers. In *2014 19th International Conference on Engineering of Complex Computer Systems*, pages 154–157. IEEE.

SecurityIdiots (2012). SQL Injection Ninja Testing Labs. http://leettime.net/sqlninja.com/index.php.

Veracode (2019). State of Software Security. https://www.thehaguesecuritydelta.com/media/com_hsd/report/219/document/state-of-software-security-2018-veracode-report.pdf.

Wassermann, G. and Su, Z. (2004). An analysis framework for security in web applications. Citeseer.

WhiteHat (2019). Top 10 Application Security Vulnerabilities of 2018. https://www.whitehatsec.com/blog/whitehat-security-top-10-application-security-vulnerabilities-of-2018.

Wu, H., Gao, G., et al. (2011). Test sql injection vulnerabilities in web applications based on structure matching. In *Proceedings of 2011 International Conference on Computer Science and Network Technology*, volume 2, pages 935–938. IEEE.

XSS (2019). XSS Game. https://xss-game.appspot.com/.

ZAkamai (2019). Web Attacks and Gaming Abuse. https://www.akamai.com/us/en/multimedia/documents/state-of-the-internet/soti-security-web-attacks-and-gaming-abuse-report-2019.pdf/.