SQL INJECTIONS

**Report on Information Security attacks andvulnerability scenario**

KESHVEE JAIN RA2011030010171

SQL injections are considered one of the most serious vulnerabilities. The adversaries inject (legitimate) SQL commands to the input data of a given application for modifying the initial SQL command, and gain access or modify private information. The adversaries exploit the fact that database makes no differentiation between end-users’ actual data and SQL commands. We have observed that about 36% of the traffic in our campus network, analyzed through deep content inspection, contains SQL injection keywords. SQL injections exploit security vulnerabilities in an application’s software. In data-driven applications, malicious SQL statements are inserted into an input field for execution (e.g. to dump the database contents to the attacker).

SQL injection can be used to attack SQL databases in a variety of ways. SQL Injection attacks are unfortunately very common, and this is due to two factors: the prevalence of SQL Injection vulnerabilities and the attractiveness of the target (databases containing the interesting/critical data for the application). The cause of SQL injection is accepting data from untrusted sources (Internet users) by application software that fails to validate and sanitize data. Subsequently, the app uses the data to dynamically construct a SQL query to the database backing the app.

SQL injection attacks cause the breach of confidentiality in the data stored in the compromised databases, which results in financial costs for recovery, downtime, regulatory penalties, and negative publicity. Any data that is passed from the user to the vulnerable web application and then processed by the supporting database represents a potential attack vector for SQL injection.

The attack vectors are supplied through HTTP GET, HTTP Post, and HTTP cookie data. Defensive programming such as employing sanitization techniques on end-users’ inputs is the obvious approach to address SQL injection attacks. SQL injection attacks can be prevented by accepting certain characters in input fields, and limiting the length of those fields. So for the username and password, for example, only accept letters for the alphabet and numbers, and limit the field to 15 characters if possible. There are multiple rules for detecting the attack depending upon the organization’s level of sensitivity. If an organization wishes to detect each and every possible

SQL Injection attack, then they simply need to watch out for any occurrence of SQL meta-characters such as the single-quote, semi-colon or double-dash. Penetration testing tool automates the process of detecting SQL injection flaws. It also includes a number of features for further exploitation of vulnerable systems, including database fingerprinting, collecting data from compromised databases, accessing the underlying file system of the server, and executing commands on the operating system via out-of-band connections.

An attacker can inject SQL into input taken from a form, as well as through the fields of a HTTP cookie. The input validation logic should consider each and every type of input that originates from the user – be it form fields or cookie information – as suspect. Also if you discover too many alerts coming in from a signature that looks out for a single-quote or a semi-colon, it just might be that one or more of these characters are valid inputs in cookies created by the Web application. Therefore, each of these signatures needs to be evaluated for the particular Web application A web application firewall (WAF) is an appliance, server plugin, or filter that applies a set of rules to an HTTP conversation.

Generally, these rules cover common attacks such as cross-site scripting (XSS) and SQL 2 injection. By customizing the rules to your application, many attacks can be identified and blocked. The effort to perform this customization can be significant and needs to be maintained as the application is modified

I have found a SQL Injection Union Based on https://intensedebate.com/commenthistory/$YourSiteId The $YourSiteId into the url is vulnerable to SQL Injection.

# Steps to reproduce

1. Logging into https://intensedebate.com
2. After create your own site on https://intensedebate.com/install and follow all steps

Now you need to know your site id, to get then you need go to https://intensedebate.com/user-dashboard and you can see on the right side of the page your site list, choice your site and click to the link Overview.

1. You will be redirected to https://intensedebate.com/dash/$YourSiteId.
2. Now you have your site id, go to the vulnerable URL with your site id https://intensedebate.com/commenthistory/$YourSiteId.
3. Now Trigger the SQL Injection with this following link https://intensedebate.com/commenthistory/$YourSiteId%20union%20select%201

,2,@@VERSION%23 (!) You need to do this with your own site id (!)

1. Now you can see 10.1.32-MariaDB on the page.

SQL Injection attacks are carried out by passing specially-formatted strings as input. In a successful attack, those special strings are passed along to a database to either execute arbitrary code or cause the server to return unanticipated results. For example, if we have a python program using [pyodbc](http://code.google.com/p/pyodbc/) which concatenates user input into a SQL query like this:

1

2

3

4

5

6

7

8

9

10

erInput = "'; Drop Table Test; --"

nn = pyodbc.connect(connString)

rs = conn.cursor()

l = ("Select City, State from dbo.ZipCodes where zipcode = '"

+

userInput +"'"

)

rs.execute(sql)

nn.commit()

Then a malicious user who carefully formats the zipcode entry could execute

unintended SQL commands. For instance, if the user provided:

1

Drop Table Test; --

Then the pro

fi

ler would show that the server would receive:

1

elect City, State from dbo.ZipCodes where zipcode = ''; Drop Table Test; --'

Assuming the program had the proper permissions, the server would obediently drop

the test table.

# Basic Techniques to Prevent SQL Injection

## Parameterize all Queries

The first, best, line of defense against SQL Injection is to parameterize all SQL queries in code. If the previous example using **pyodbc** had been parameterized, it could look like:

1

2

3

4

5

6

7

8

9

serInput = "'; Drop Table Test; --"

onn = pyodbc.connect(connString)

urs = conn.cursor()

ql = "Select City, State from dbo.ZipCodes where zipcode = ?"

urs.execute(sql, (userInput,))

onn.commit()

This causes the pro

fi

ler to receive quite a few messages, but the key part is:

1

xec sp\_prepexec @p1 output,N'@P1 varchar(22)',N'Select City, State from

bo.ZipCodes where zipcode = @P1','''; Drop Table Test; --'

Since it received the malicious code as a variable, the server would simply look for the value in the table and return a blank result sets. The malicious string is never executed, so the test table is never dropped.

Similarly, most ORMs like [SQLAlchemy](http://www.sqlalchemy.org/) will automatically parameterize all SQL statements under normal circumstances. Thus, they provide a good initial defense against SQL injection.

# Use Only Stored Procedures

The use of stored procedures by themselves [does not](http://www.simple-talk.com/sql/t-sql-programming/to-sp-or-not-to-sp-in-sql-server/) [provide direct protection against SQL injection](http://www.simple-talk.com/sql/t-sql-programming/to-sp-or-not-to-sp-in-sql-server/), although it can properly be used as part of a [more comprehensive defense](http://www.dotnetmonster.com/Uwe/DirItem.aspx/Articles/NET-Framework/ADO-NET/Beyond-Stored-Procedures). To see why stored procedures cannot by themselves protect against SQL injection, consider one that queries to retrieve the city and state for a zip code like:

1

2

3

4

5

6

7

reate procedure dbo.GetCityState

@zipcode varchar(15)

s

elect city, State

rom dbo.ZipCodes

here zipcode = @zipcode

Then if there is a python program that executes:

1

2

3

4

ql = "exec dbo.GetCityState '" + userInput + "'"

urs.execute(sql)

onn.commit()

Then an attacker might provide:

1

Drop Table Test; --

As the input, which will send…

1

xec dbo.GetCityState ''; Drop Table Test; --'

…to the server. Which, again assuming proper permissions, would drop the Test table just as it did without the stored procedure.

Of course, that type of attack can be prevented by parameterizing the input, just as with the standard select statement. However, if the stored procedure itself uses dynamic SQL that is made through concatentation, then the stored procedure may execute the malicious commands even if the calling program properly parameterizes. This can be prevented by parameterizing dynamic SQL in stored procedures through sp\_executesql and is discussed in “[The Curse and Blessings of Dynamic](http://www.sommarskog.se/dynamic_sql.html) [SQL](http://www.sommarskog.se/dynamic_sql.html)“.

The greatest value for using stored procedures in preventing SQL injection is that the DBA can set permissions for the application account so that its only way to interact with the SQL Server is through stored procedures. (See [**SQL Server Security Workbench Part**](http://www.simple-talk.com/sql/t-sql-programming/sql-server-security-workbench-part-1/)

[**1**](http://www.simple-talk.com/sql/t-sql-programming/sql-server-security-workbench-part-1/) **)** This would mean that most SQL injection attacks would fail due to lack of permissions even if the calling program did not parameterize. This of course still leaves open the possibility of SQL injection working through dynamic SQL inside the stored procedures, but the stored procedures can be given an “execute as” clause which limits their permissions to only those needed by the procedure. It is generally easier to verify that all stored procedures are written to guard against SQL injection then it is to check every place where the application interacts with the SQL Server.

# Limiting Permissions

This naturally leads to a very effective method of preventing some attacks and limiting the damages from SQL injection attacks, namely using the account with the lowest permissions possible for a job. If the account being used does not have permission to drop a table, then it will not be dropped even if the command is slipped to SQL Server. Similarly, if the account has only read access, an attacker might be able to gain some information, which can certainly cause problems, but the attacker will not be able to modify or destroy the data, which is frequently worse. Even read permissions can be strictly limited in SQL server, to limit which tables can be viewed. If the application only needs selected columns from a table, then read permission on the view can be granted rather than the full table.

# Validating input

User input should always be treated with care and there are a number of reasons to validate all the user input before further processing. Validation code can also help to avoid wasting server resources by restricting requests that would not return useful results and they can provide much more helpful messages to the user than a SQL error message or empty result set would likely provide. They can also help stop SQL Injection by rejecting, outright, any forms of input that could be used to perform a SQL injection.

Because of its many advantages, it is always important to validate user input, but it is particularly significant when the user input is being passed on to other routines for further processing, or in some of the rare cases where it is impossible to fully parameterize the input. For instance, if you are dealing with the rare situation where the users are required to provide a table name for a DDL statement, the table name cannot be passed in as a parameter and must be concatenated at some point. In that situation, validation of the input is a crucial defense against injection attacks. Similarly, if the input is passed in to a stored procedure, then it is possible that the stored procedure will use it to generate dynamic SQL via concatenation, even if the program properly parameterizes the procedure call. With the benefits that validation can bring, it is generally wise to validate all user input, even when fully parameterizing database calls and using an account with limited permissions.

# Concealing Error Messages

Injection attacks often depend on the attacker having at least some information about the database schema. He often gains this through trial and error, and the error messages will tell the attacker quite a lot about the schema. Both SQL Server and python generally provide clear, informative error messages that are incredibly helpful to programmers, but can also provide information to a malicious user. **Pyodbc**, in particular, will normally raise a **pyodbc.ProgrammingError** exception, which helpfully includes the SQL Server error message.

Encasing a python call to SQL Server in a try/except block will enable the program to provide a more user friendly error message, which does not contain useful information for attackers, to the end user. If used along with something like [sys.exc\_info](http://docs.python.org/library/sys.html) and [a logging package](http://docs.python.org/library/logging.html), the **except** blocks can log all errors for later analysis while displaying a user friendly message to the end user. A very basic example might look like:

0

1

3

port logging

port sys

gFile = 'test.log'

gging.basicConfig(filename=logFile,lev

=

)

logging.DEBUG

stablish connection, create faulty

L, removed for brevity

y:

curs.execute(sql)

conn.commit()

cept:

print 'User Friendly Error'

logging.debug(sys.exc\_info())

Of course, to ensure no unfiltered messages get through it is possible to override [the standard exception hook](http://docs.python.org/library/sys.html#sys.excepthook) like:

1

2

3

4

5

6

7

8

9

mport sys

ef new\_exceptionhandler(type, value, tb):

#put in custom logging code here

print 'There was a general error.'

ys.excepthook = new\_exceptionhandler

rint aVariable #NameError since aVariable is not defined

This will not have any impact on exception handling code, but it will prevent standard

error messages from reaching a user for unhandled exceptions.

# Limiting Damage

As well as taking steps to prevent attacks like SQL injection, there are other general security steps that can be taken to limit the damage. Limiting the permissions of the accounts used has been mentioned, as it can stop many attacks outright, but it will also limit the amount of damage that can be done by a successful attack. But there are other methods that can help mitigate the damage done by an attack.

## Use encryption/hash functions where appropriate

When data is properly encrypted, it can be made of little value to someone without an encryption key. Cell level encryption can assist with protecting against unauthorized access to sensitive data SQL Server has supported it since SQL Server 2005. Transparent Data Encryption (TDE), while useful for protecting the database against other forms of attack, is of very little value against SQL injection.

Passwords in particular should not be stored in clear text, and hashing is generally better than encrypting as it makes it harder to recover the original plaintext. Of course, even hashing may provide only limited security if it is not handled properly. For example, rainbow tables (http://en.wikipedia.org/wiki/Rainbow\_table) exist for the MD5 algorithm. They make it relatively practical to determine the original plaintext from a single iteration of MD5 without salting. There are libraries that can make hashing and salting a password relatively easy in python, including [hashlib](http://docs.python.org/library/hashlib.html).

## Segregate data

Segregating data into different systems, depending on the level of security it needs, can help limit the reach of an attack. It often even makes sense to ensure that truly sensitive data is stored in a way that is not accessible from an outside network. This helps to ensure that even if an attacker compromises a system, that it will not immediately lead to the attacker compromising all systems. Of course, it is necessary to ensure that the same log on credentials are not shared between the segregated systems, otherwise if one set of log on credentials is compromised in some way it may lead directly to compromising other systems.

## Auditing and Logging

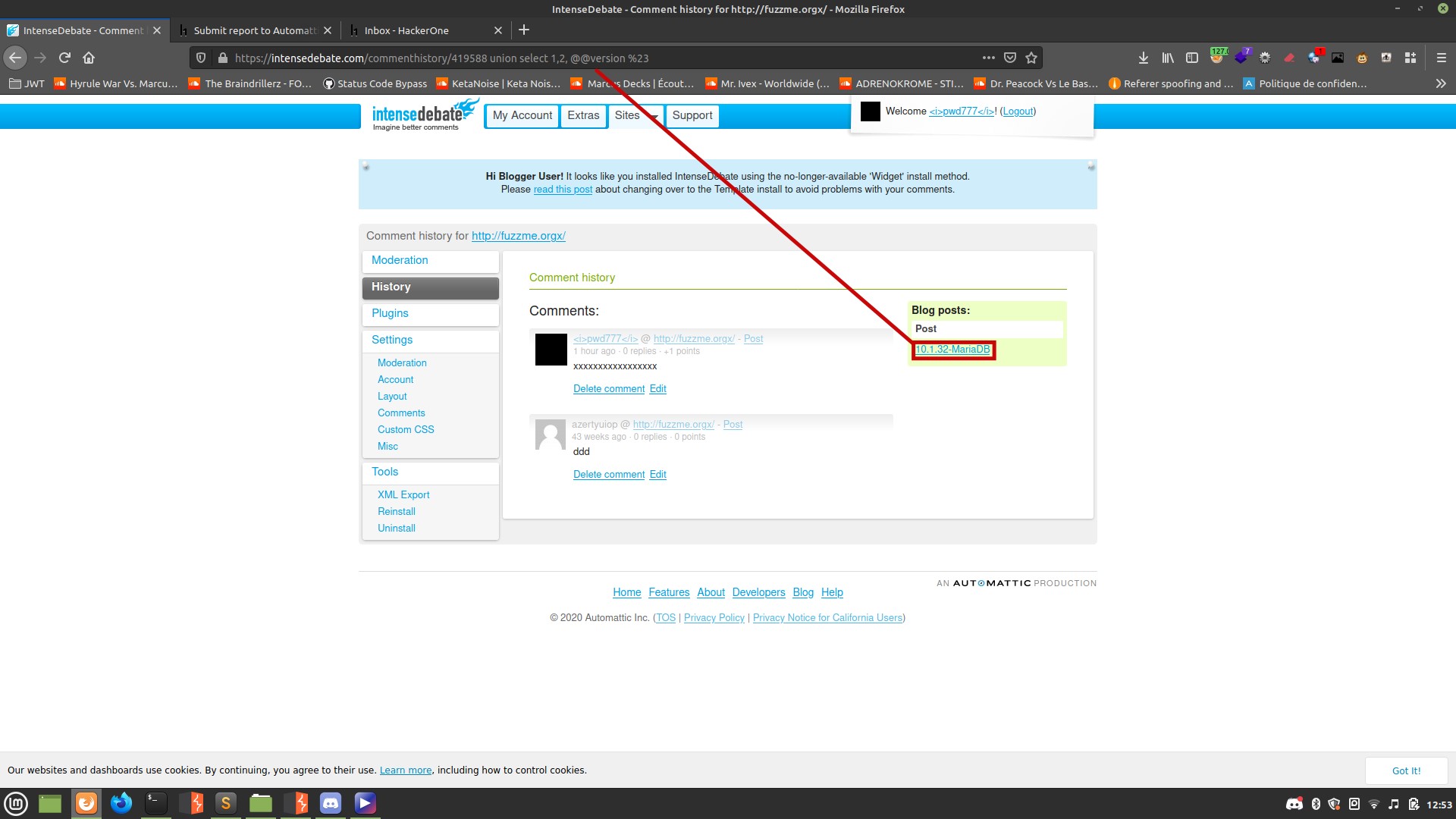
Auditing and logging will never help prevent SQL injection or any other attack. However, it is likely to help detect the attacks, and may help in recovering from it. There are a number of tools within SQL Server such as [Change Data](http://www.simple-talk.com/sql/learn-sql-server/introduction-to-change-data-capture-(cdc)-in-sql-server-2008/) [Capture](http://www.simple-talk.com/sql/learn-sql-server/introduction-to-change-data-capture-(cdc)-in-sql-server-2008/) and [SQL Server Audit](http://www.simple-talk.com/sql/database-administration/sql-server-audit-magic-without-a-wizard/). Custom written triggers could also be used to monitor and log changes. There are also a number of external tools that can provide more options such as [SQL Server Compare and SQL Server Data Compare](http://www.red-gate.com/products/sql-development/sql-compare/learn-more/audit-trail). The Logging library and other similar libraries make logging from the python application also relatively easy.

current\_user()

Image F1096976: currentUser.png

193.59

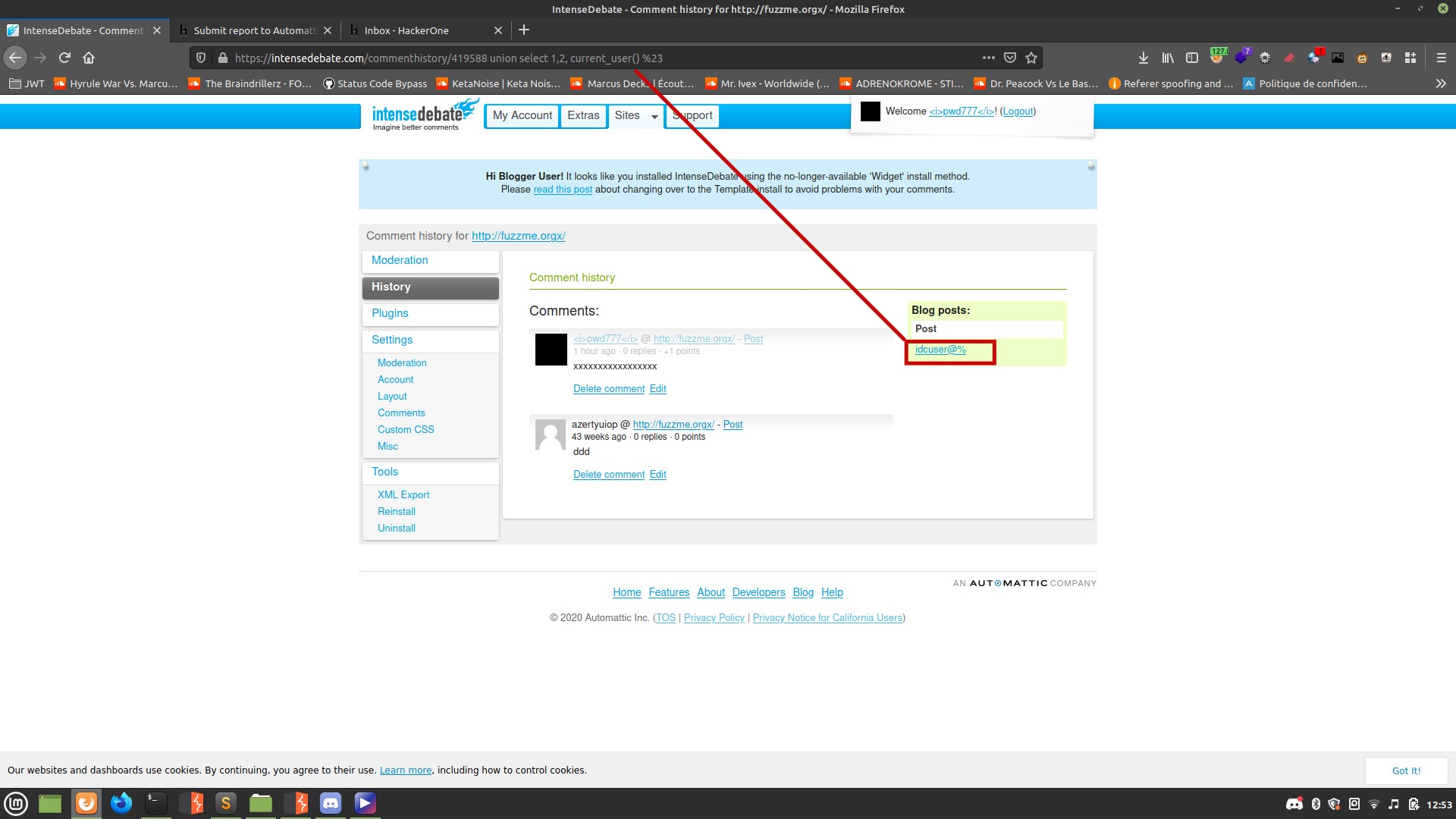
KiB



[Zoom in](https://hackerone.com/reports/1046084) [Zoom out](https://hackerone.com/reports/1046084) [Cop](https://hackerone.com/reports/1046084)[y](https://hackerone-us-west-2-production-attachments.s3.us-west-2.amazonaws.com/fa3kixvecnjeelb0g2m5tzbeei5w?response-content-disposition=attachment%3B%20filename%3D%22currentUser.png%22%3B%20filename%2A%3DUTF-8%27%27currentUser.png&response-content-type=image%2Fpng&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=ASIAQGK6FURQRHNSKARN%2F20221113%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20221113T152336Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEMb%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJGMEQCIBlzmAWLhJPcdujJkHZpHLLngm7GXvtMC0O%2B9UWWLaRhAiA%2Fu3mj3YI9TGM1ibAgY3BD9zxHwn37uwHE5kV%2BMgJHTirVBAi%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F8BEAMaDDAxMzYxOTI3NDg0OSIMa3lL774lAFksiY%2BYKqkEzcSRUsJBqx2IOtCkGYH83lT7jIV8ntc9dbM0W7ydFSgDIX5dsGOf9mTx9kOHLIdG7fOWMlgszbcO4%2FAcbT8%2FmJg4AyEkuT63qBHRMlFo1HVj0vkOgrfjpB09fy1SrpGqtWe3rjN0FTiMpCxk0MOs3z%2BR9h%2BDM5WShaYnPf9pcYLP0SYMk%2F2cXuIUVnL13EXGzxGglGbD3cH%2BaGBGpQH4u29wpnf%2FvV8KAC70ZliSgTN6q7WP9wCbBhvOqlzL0pbIEc8rUhZneZNz%2FwsCIznrAntRTlRDKtn9aok4J1wos%2BzgedsEskQVMX1%2FYziBAm1cXDkPoKqCdNY7Mehs%2FS3zaYZVfMkvch0ZnL320%2FDfCpcjUfhSanobHQEbfeCr1s00mpj6TfF%2F87zGUL%2BZB88RZB1NW9l36gItUnCNfXcU8wfZ1NQW%2BtxYm2FfH19wOiaWc7B72v9ogDDkaf7SbZNj5lh2NemEAwH5oPrtFh98syCwGeB94dVVua3MPs4M2nJ0RftXivNWpbwrD7xbLLs5Evn11%2FtcnBNvx%2Bh8MIk5IPY1Y7YQfcczhy46sjOTdSjXdJDOhaxOx2IaqbrDqquojU4xh6tDcy8I8s5pODQMPB6TMLyt6SzZRmZHi3RWdb9w6Peinnl0bU3GSc%2BzNkrKz0yRYvkHO%2FdZ2ev1YBEqr0J3h4BhRQhbH1sbC6wRgUdB8aSX%2FbnQs3xud1MR3Zn8r50tVX1BOJm05jDs%2BMObBjqqAZd5jvqqJFl023B7EczCHgPlFbCumqoIXdhJyW75qRBf6xrs9r%2BMwQ21eef1aFkaL%2FcMHRL9HqnDLGRhphByepkFyEA8VnWlOB1OIxm%2FRPeZ65EutjJ4UB9yEb30CKaT22uMGr61SjrvnjrGMEKMZo7c4%2FaPsLs7aXz97J8ZbnMhEE8CJ1IT%2Bfz3BRJeh69Lg3eZdzraaNC7olw%2FJdFLSUtsDrs%2BL%2B48%2Bmqm&X-Amz-SignedHeaders=host&X-Amz-Signature=8e6e5b879c31da1f8035425e90562b8d10aaab729ee987f0743f602f32ea6196) [Download](https://hackerone-us-west-2-production-attachments.s3.us-west-2.amazonaws.com/fa3kixvecnjeelb0g2m5tzbeei5w?response-content-disposition=attachment%3B%20filename%3D%22currentUser.png%22%3B%20filename%2A%3DUTF-8%27%27currentUser.png&response-content-type=image%2Fpng&X-Amz-Algorithm=AWS4-HMAC-SHA256&X-Amz-Credential=ASIAQGK6FURQRHNSKARN%2F20221113%2Fus-west-2%2Fs3%2Faws4_request&X-Amz-Date=20221113T152336Z&X-Amz-Expires=3600&X-Amz-Security-Token=IQoJb3JpZ2luX2VjEMb%2F%2F%2F%2F%2F%2F%2F%2F%2F%2FwEaCXVzLXdlc3QtMiJGMEQCIBlzmAWLhJPcdujJkHZpHLLngm7GXvtMC0O%2B9UWWLaRhAiA%2Fu3mj3YI9TGM1ibAgY3BD9zxHwn37uwHE5kV%2BMgJHTirVBAi%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F%2F8BEAMaDDAxMzYxOTI3NDg0OSIMa3lL774lAFksiY%2BYKqkEzcSRUsJBqx2IOtCkGYH83lT7jIV8ntc9dbM0W7ydFSgDIX5dsGOf9mTx9kOHLIdG7fOWMlgszbcO4%2FAcbT8%2FmJg4AyEkuT63qBHRMlFo1HVj0vkOgrfjpB09fy1SrpGqtWe3rjN0FTiMpCxk0MOs3z%2BR9h%2BDM5WShaYnPf9pcYLP0SYMk%2F2cXuIUVnL13EXGzxGglGbD3cH%2BaGBGpQH4u29wpnf%2FvV8KAC70ZliSgTN6q7WP9wCbBhvOqlzL0pbIEc8rUhZneZNz%2FwsCIznrAntRTlRDKtn9aok4J1wos%2BzgedsEskQVMX1%2FYziBAm1cXDkPoKqCdNY7Mehs%2FS3zaYZVfMkvch0ZnL320%2FDfCpcjUfhSanobHQEbfeCr1s00mpj6TfF%2F87zGUL%2BZB88RZB1NW9l36gItUnCNfXcU8wfZ1NQW%2BtxYm2FfH19wOiaWc7B72v9ogDDkaf7SbZNj5lh2NemEAwH5oPrtFh98syCwGeB94dVVua3MPs4M2nJ0RftXivNWpbwrD7xbLLs5Evn11%2FtcnBNvx%2Bh8MIk5IPY1Y7YQfcczhy46sjOTdSjXdJDOhaxOx2IaqbrDqquojU4xh6tDcy8I8s5pODQMPB6TMLyt6SzZRmZHi3RWdb9w6Peinnl0bU3GSc%2BzNkrKz0yRYvkHO%2FdZ2ev1YBEqr0J3h4BhRQhbH1sbC6wRgUdB8aSX%2FbnQs3xud1MR3Zn8r50tVX1BOJm05jDs%2BMObBjqqAZd5jvqqJFl023B7EczCHgPlFbCumqoIXdhJyW75qRBf6xrs9r%2BMwQ21eef1aFkaL%2FcMHRL9HqnDLGRhphByepkFyEA8VnWlOB1OIxm%2FRPeZ65EutjJ4UB9yEb30CKaT22uMGr61SjrvnjrGMEKMZo7c4%2FaPsLs7aXz97J8ZbnMhEE8CJ1IT%2Bfz3BRJeh69Lg3eZdzraaNC7olw%2FJdFLSUtsDrs%2BL%2B48%2Bmqm&X-Amz-SignedHeaders=host&X-Amz-Signature=8e6e5b879c31da1f8035425e90562b8d10aaab729ee987f0743f602f32ea6196)

Video POC

s



### Impact

Full database access holding private user information and Reflected

Cross-Site-Scripting