Ensuring Authorized Updates in Multi-user Database-Backed Applications

Kevin Eykholt, Atul Prakash, Barzan Mozafari

University of Michigan Ann Arbor
Database Backend

- Web Applications allow users to remotely access services
- Information stored in database
Symantec Patches High Risk Vulnerabilities in Endpoint Protection

By SecurityWeek News on March 21, 2016

Symantec has released an update for its Symantec Endpoint Protection (SEP) to resolve three high risk security vulnerabilities in the product.

According to an advisory issued Mar. 17, the security flaws in Symantec Endpoint Protection could potentially result in authorized users with low privileges gaining elevated access to the Management Console. Moreover, the security firm warns that SEP Client security mitigations could be bypassed to achieve arbitrary code execution on a targeted client.

The first of the three security issues is a cross-site request forgery vulnerability in the management console for SEPM (CVE-2015-8152), caused by an insufficient security check in SEPM. An authorized but less-privileged user could gain unauthorized elevated access to the SEPM management console by including arbitrary code in authorized logging scripts.

In addition to the CSRF issue, Symantec resolved an SQL injection vulnerability in SEPM (CVE-2015-8153). This security flaw can also be exploited by an authorized, logged-in user to potentially elevate access to administrative level on the application.
Built-in Database Access Controls

Some DBMS provide fine-grained access control based on context of connected user.
Built-in Database Access Controls

But database user is not the same as application user!
Application based Access Controls

• CLAMP and Nemesis both define per-user access control policies on each table

• Most existing work defines access control policies using database views


Database View

• Database views restrict a user to a portion of the database using a SELECT query
• “Only allow customers to view their own orders”

```sql
SELECT O.*
FROM orders O
WHERE O.cust_id = $current_id
```
Database View

• Existing techniques use views to restrict read/write access
• The same query can also express the write policy: “Only allow customers to update their own orders”

```
SELECT O.*
FROM orders O
WHERE O.cust_id = $current_id
```
What if you can’t map a user to authorized rows in the table?
Problem

• “Customers can only leave reviews for items they have purchased”

```sql
SELECT R.*
FROM reviews R, orders_products P, orders O
WHERE O.cust_id = current_id AND
  O.oid = OP.oid AND
  OP.product_id = R.product_id AND
  R.cust_id = $current_id
```
Survey of Existing Web Applications

<table>
<thead>
<tr>
<th>Web App</th>
<th>Total Tables</th>
<th>Tables Requiring Join Policy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wordpress</td>
<td>12</td>
<td>4 (33%)</td>
</tr>
<tr>
<td>hotCRP</td>
<td>24</td>
<td>6 (25%)</td>
</tr>
<tr>
<td>LimeSurvey</td>
<td>36</td>
<td>18 (50%)</td>
</tr>
<tr>
<td>osCommerce</td>
<td>40</td>
<td>4 (10%)</td>
</tr>
<tr>
<td>MediaWiki</td>
<td>48</td>
<td>10 (21%)</td>
</tr>
<tr>
<td>WeBid</td>
<td>55</td>
<td>5 (9%)</td>
</tr>
<tr>
<td>Drupal</td>
<td>60</td>
<td>12 (20%)</td>
</tr>
<tr>
<td>myBB</td>
<td>75</td>
<td>8 (11%)</td>
</tr>
<tr>
<td>ZenCart</td>
<td>96</td>
<td>18 (19%)</td>
</tr>
<tr>
<td>Cyclos</td>
<td>185</td>
<td>24 (13%)</td>
</tr>
<tr>
<td><strong>Average Percent</strong></td>
<td><strong>21%</strong></td>
<td></td>
</tr>
</tbody>
</table>
Insecure Code

Symantec has released an update for its Symantec Endpoint Protection (SEP) to resolve three high risk security vulnerabilities in the product.

According to an advisory issued Mar. 17, the security flaws in Symantec Endpoint Protection could potentially result in authorized users with low privileges gaining elevated access to the Management Console. Moreover, the security firm warns that SEP Client security mitigations could be bypassed to achieve arbitrary code execution on a targeted client.

The first of the three security issues is a cross-site request forgery vulnerability in the management console for SEPM (CVE-2015-8152), caused by an insufficient security check in SEPM. An authorized but less-privileged user could gain unauthorized elevated access to the SEPM management console by including arbitrary code in authorized logging scripts.

In addition to the CSRF issue, Symantec resolved an SQL injection vulnerability in SEPM (CVE-2015-8153). This security flaw can also be exploited by an authorized, logged-in user to potentially elevate access to administrative level on the application.

Symantec patches high risk vulnerabilities in endpoint protection.  
Design Goals

• Generality: Access control policy can be enforced read/write queries
• Correctness: Current user can only view/modify authorized information
• Architectural Compatibility:
  – Solution works with existing web applications without requiring major changes
  – Solution is not database specific
• Simple: Preliminary knowledge overhead is low
Proposed Solution

Implement the control in the database driver
Solution Outline

• **What is Query Safety?**
• How can Query Safety be enforced?
• Experiments with proposed methodology
Database Policies

• Two types of policies:
  – Read Policy
  – Write Policy

• A policy is composed of a set of rules for each database table
Policy Definitions

• Customers can only view/modify their own orders

Read/Write Policy

SELECT 0.*
FROM orders 0
WHERE 0.cust_id = current_id
Policy Definitions

• Customers can view items available in the store

Read Policy

```sql
SELECT P.*
FROM products P
```

Write Policy

```sql
SELECT P.*
FROM products P
WHERE 1=0
```
Policy Definitions

• Customers can read any review, but only leave reviews for items they purchased

Read Policy

SELECT R.*
FROM reviews R

Write Policy

SELECT R.*
FROM reviews R, orders_products P, orders O
WHERE O.cust_id = current_id AND
  O.oid = OP.oid AND
  OP.product_id = R.product_id AND
  R.cust_id = current_id
Query Safety

- A **safe** query is one that allows a user to only view/modify authorized tuples in the database identified by the security policy
  - A read query is safe if it is **read-safe**
  - A write query is safe if it is **write-safe**
Read-Safe Query

• A query is **read-safe** if the query’s result is unchanged when executed on only the tables a user is authorized to access.
Read-Safe Query

Read Policy

```sql
SELECT O.*
FROM orders O
WHERE O.cust_id = 1
```

Read-Safe

```sql
SELECT orders_id
FROM orders
WHERE cust_id = 1
```

Not Read-Safe

```sql
SELECT *
FROM orders
```
Write-Safe Query

• A query is **write-safe** if:
  1. The query is read-safe
  2. The query does not modify unauthorized tuples
     • The results of the query should not change if the query is restricted to modifying tuples in the write policy
Write-Safe Query

Write Policy

SELECT O.*
FROM orders O
WHERE O.cust_id = 1

Write-Safe Queries

DELETE FROM orders
WHERE cust_id = 1

Not Write-Safe Queries

DELETE FROM orders

INSERT INTO orders
AS SELECT *
FROM orders
Solution Outline

• What is Query Safety?
• **How can Query Safety be enforced?**
• Experiments with proposed methodology
Ensuring Query Safety

Query -> Read Set Intersection

Read Set Intersection -> Read Safe Query?

Read Safe Query? -> Select Query?

Select Query? -> Execute Safe Query

Select Query? -> Phantom Extraction

Phantom Extraction -> Write Safe Query

Execute Safe Query
Read Set Intersection

• Any query can be transformed to a read-safe query by adding additional conditions to the WHERE clause of any SELECT queries

Read Policy

```
SELECT 0.*
FROM orders 0
WHERE 0.cust_id = 1
```
Phantom Extraction

1. Transform the query into a read-safe query (Read Set Intersection)
2. Modify the resulting query to only update tuples authorized by the write policy

• Two strategies for step 2:
  – V-Copy
  – No-Copy
V-Copy

- Determines query safety using temporary tables

- Given a query, V-Copy always results in safe behavior
**V-Copy**

1. Copy schema of T
2. Execute the query on the copy
3. Check if result is allowed by the write policy
4. Execute Null query

```
Customer 1

<table>
<thead>
<tr>
<th>oid</th>
<th>cust_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
</tr>
</tbody>
</table>

Driver

```INSERT INTO orders (oid, cust_id)
VALUES(22, 2);
```

```
<table>
<thead>
<tr>
<th>oid</th>
<th>cust_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>2</td>
</tr>
</tbody>
</table>

SELECT O.*
FROM orders O
WHERE O.cust_id = 1
```
V-Copy

1. Copy authorized write tuples from T
2. Execute the query on the copy
3. Check if result is allowed by the write policy
4. Propagate changes

```
SELECT O.*
FROM orders O
WHERE O.cust_id = 1
```
No-Copy

• Can only use when:
  – Write Policy for table does not contain a join
  – Query is not a nested INSERT
  – And if the query is UPDATE, the SET clause only contains static values
No-Copy

* If the query is DELETE, append additional conditions to WHERE clause

**Write Policy**

```sql
SELECT O.*
FROM orders O
WHERE O.cust_id = 1
```

DELETE
FROM orders

DELETE
FROM orders
WHERE O.cust_id = 1
Solution Outline

• What is Query Safety?
• How can Query Safety be enforced?
• **Experiments with proposed methodology**
SafeD

- We created SafeD, a custom JDBC driver, that implements both V-Copy and No-Copy
Benchmark

• TPC-C Benchmark
  – Provided in OLTPBenchmark
  – 5 transaction types
  – Scale factor: 20
  – Worker count:
    • 60 (PostgreSQL)
    • 100 (MySQL)
  – Phase Duration: 10 minutes
Access Roles and Policy

- **Customer**: Executes new order, order status, payment transactions
- **Managers**: Executes delivery and stock transactions

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Customer</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer(C_ID, C_D_ID, C_W_ID)</td>
<td>((\text{CID, DID, WID}))</td>
<td>Full Access</td>
</tr>
<tr>
<td>District(D_ID, D_W_ID)</td>
<td>((\text{DID, WID}))</td>
<td>((\text{DID, WID}))</td>
</tr>
<tr>
<td>Warehouse(W_ID)</td>
<td>((\text{WID}))</td>
<td>Full Access</td>
</tr>
<tr>
<td>OOrder(O_C_ID, O_D_ID, O_W_ID)</td>
<td>((\text{CID, DID, WID}))</td>
<td>((\text{DID, WID}))</td>
</tr>
<tr>
<td>New_Order(NO_O_ID)</td>
<td>Contain (OID) in OOrder</td>
<td>Full Access</td>
</tr>
<tr>
<td>Order_Line(OL_O_ID)</td>
<td>Contain (OID) in OOrder</td>
<td>Full Access</td>
</tr>
<tr>
<td>History(H_C_ID, H_D_ID, H_W_ID)</td>
<td>((\text{CID, DID, WID}))</td>
<td>Full Access</td>
</tr>
<tr>
<td>Item</td>
<td>Full Access</td>
<td>Full Access</td>
</tr>
<tr>
<td>Stock</td>
<td>No Access</td>
<td>Full Access</td>
</tr>
</tbody>
</table>
Performance Measures

• Average Latency

\[ AL_{S,r} = \frac{\sum_{i=1}^{N} L_{i}^{S,r}}{N} \]

• Average Throughput

\[ AT_{n,S,r} = \frac{\sum_{i=1}^{N} T_{i}^{S,r}}{N} \]
Performance Measurements

• When queries in the workload are safe
  – What is the performance overhead compared to a database without built-in access controls?
  – How does SafeD compare to an existing built-in access control mechanism

• How does overall performance vary as the number of unsafe queries in the workload increase?
MySQL Performance Results

No-Copy has an average latency overhead of 5.9%
V-Copy has an average latency overhead of 6.1%
PostgreSQL Performance Results

SafeD access controls has comparable performance to built-in Postgres Access Control
## Modified Security Policy

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Customer</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer(C_ID, C_D_ID, C_W_ID)</td>
<td>(CID, DID, WID)</td>
<td>Full Access</td>
</tr>
<tr>
<td>District(D_ID, D_W_ID)</td>
<td>(DID, WID)</td>
<td>(DID, WID)</td>
</tr>
<tr>
<td>Warehouse(W_ID)</td>
<td>(WID)</td>
<td>Full Access</td>
</tr>
<tr>
<td>OOrder(O_C_ID, O_D_ID, O_W_ID)</td>
<td>(CID, DID, WID)</td>
<td>(DID, WID)</td>
</tr>
<tr>
<td>New_Order(NO_O_ID)</td>
<td>Contain (OID) in OOrder</td>
<td>Full Access</td>
</tr>
<tr>
<td>Order_Line(OL_O_ID)</td>
<td>Contain (OID) in OOrder</td>
<td>Full Access</td>
</tr>
<tr>
<td>History(H_C_ID, H_D_ID, H_W_ID)</td>
<td>(CID, DID, WID)</td>
<td>Full Access</td>
</tr>
<tr>
<td>Item</td>
<td>Full Access</td>
<td>Full Access</td>
</tr>
<tr>
<td>Stock</td>
<td>No Access</td>
<td>Full Access</td>
</tr>
</tbody>
</table>
Security Policy – Policy 2

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Customer</th>
<th>Manager</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer(C_ID, C_D_ID, C_W_ID)</td>
<td>=(CID,DID,WID)</td>
<td>Full Access</td>
</tr>
<tr>
<td>District(D_ID, D_W_ID)</td>
<td>=(DID,WID)</td>
<td>=(DID,WID)</td>
</tr>
<tr>
<td>Warehouse(W_ID)</td>
<td>=(WID)</td>
<td>Full Access</td>
</tr>
<tr>
<td>OOrder(O_C_ID, O_D_ID, O_W_ID)</td>
<td>=(CID,DID,WID)</td>
<td></td>
</tr>
<tr>
<td>New_Order(NO_O_ID)</td>
<td>Contain (OID) in OOrder</td>
<td></td>
</tr>
<tr>
<td>Order_line(OL_O_ID)</td>
<td>Contain (OID) in OOrder</td>
<td>Full Access</td>
</tr>
<tr>
<td>History(H_C_ID, H_D_ID, H_W_ID)</td>
<td>=(CID,DID,WID)</td>
<td>Full Access</td>
</tr>
<tr>
<td>Item</td>
<td>Full Access</td>
<td>Full Access</td>
</tr>
<tr>
<td>Stock</td>
<td>No Access</td>
<td>Full Access</td>
</tr>
</tbody>
</table>

\[
O_{C_ID} > 0
\]

Contain (OID) in OOrder
PostgreSQL Performance Results

No-Copy can sustain a much higher transaction throughput with much lower latency.
Unsafe Queries

- Previous performance numbers were measured when all the queries in the workload were safe
- In addition to the normal TPC-C queries, we added a mix of unsafe read and write queries to the workload
V-copy does not scale well as the number of unsafe queries increases.
PostgreSQL Performance Results – Unsafe Queries with Policy 1

No-Copy sustains a higher average transaction rate
Syntax Knowledge Required

- SafeD
  - SELECT
- PostgreSQL
  - Policy and policy condition
  - Role
- Oracle
  - System Context
  - Login Trigger
  - Policy
  - Policy function
Developer Effort

• SafeD only requires basic SQL syntax to define
  – Policies defined as intuitive SELECT statements

• Postgres and Oracle both have database specific syntax for access control

<table>
<thead>
<tr>
<th>Access Control Mechanism</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>SafeD</td>
<td>36</td>
</tr>
<tr>
<td>Postgres’s Built-in Access Control</td>
<td>54</td>
</tr>
<tr>
<td>Oracle’s Built-in Access Control (a.k.a. VPD)</td>
<td>544</td>
</tr>
</tbody>
</table>
Ensuring Authorized Updates in Multi-user Database-Backed Applications