**1. Authentication flags and privilege escalation**

Since applications have their own access-control lists and privileges, if the implementation of the authorization is weak, it opens up vulnerabilities that can be exploited, such as accessing another's content or becoming a higher-level user with greater permissions.

**[SM-D1-R01]** Identifying parameter names that have something to do with ACL/permission that could become a target, and the tester can use fuzzing tools to try and change bit patterns or permission flags, which may show the point at which exploitation, escalating privileges or bypassing authentication can be achieved by an attacker.

### 2. Critical parameter manipulation and access to unauthorized information/content

HTTP GET and POST requests are typically accompanied with several parameters when submitted to the application, typically in the form of name/value pairs, JSON, XML and so forth, but they can be tampered with and guessed by predicting.

**[SM-D1-R02]** Tests for this look for easily guessable values and whether a parameter's value can be changed in order to gain unauthorized access.

## *JSON XSRF Attacks*

All modern web browsers implement **Same Origin Policy (SOP)** on website content. This policy avoids one website from writing contents of frame that was issued from a different domain. This policy was implemented against Frame Injection flaws. JSON CSRF exploits SOP by lifting one way restriction on website and allows data from another website to execute on a different domain. This vulnerability is the result of flaw in SOP which treats JavaScript as code, not as data. According to browser policy, a code is allowed to be downloaded and executed over a client browser though the original source of the script is different.

**JSON** means **Java Script Object Notification** which is a data transfer format for JavaScript interpreters. It is used in AJAX based applications as an alternative to standard XML data transfer format. In these applications, requests are made using **XML Http Request** to a server and the server returns data in JSON format. The received data is transferred on to the client side. Since JavaScript is used to transmit data, then pure code SOP policy can get exploited to gain data generated by other applications. This data is transmitted back in the form of an array. Therefore, because of JSON, an XSRF attack can easily be executed on vulnerable site. It is quite clear that JSON CSRF attacks can be implemented over an AJAX based website which uses JSON data transfer format against standard XML data transfer format.

*The following are preventive measures that can be implemented against XSRF attacks.*

* *First of all, the application must implement all kinds of basic XSRF attacks.*
* *Always use unpredictable parameters for JSON objects.*
* *JSON XSRF attacks are possible because the application can send XML Http Requests to retrieve JSON data and it can only retrieve data by using the GET method; therefore, it is better to implement only the POST method as a countermeasure against JSON XSRF.*

### 3. Developer's cookie tampering and business process/logic bypass

Cookies are often used to maintain state over HTTP, but developers are not just using session cookies, but are building data internally using session-only variables. Application developers set new cookies on the browser at important junctures which exposes logic holes. The danger is that these cookies can be reverse engineered or have values that can be guessed or deciphered and attackers try to identify these holes that are easy to exploit.

**[SM-D1-R03]** Tests here typically involve analysis of cookies delivered during profiling, and looking for easily guessable values, and whether a cookie value can be changed.

### 4. LDAP parameter identification and critical infrastructure access

A common use of LDAP is to provide a central place to store usernames and passwords. This allows many different applications and services to connect to the LDAP server to validate users. This has a major benefit that allows a central place to update and change user passwords. LDAP is becoming an important aspect for large applications and may get integrated with "single sign-on" as well. Many infrastructure layer tools like SiteMinder and Load Balancer use LDAP for both authentication and authorization. LDAP parameters can carry business-logic decision flags that can be abused or leveraged. Attackers can find business-layer bypasses and logical injections if the application is not doing enough validation.

**[SM-D01-R04]** Tests for this focus on finding parameters linked with DAP, such as those taking email or usernames, which are prospective targets.

### 5. Business constraint exploitation

The application's business logic should have defined rules and constraints, but if poorly designed, attackers can crawl them and browse through hidden fields and understand their context.

**[SM-D01-R05]** Test hidden parameters and values, checking business-specific calls that can become a target and manipulated.

### 6. Business flow bypass

Applications include flows that are controlled by redirects and page transfers. However, in many cases, this flow can be bypassed, which can lead to an error condition or information leakage, which can help an attacker identify critical backend information.

**[SM-D01-R06]** Test whether business functionality and parameters can be tampered with through a proxy.

### 7. Exploiting client-side business routines embedded in JavaScript, Flash or Silverlight

Many business applications now run on rich Internet application frameworks leveraging JavaScript, and in many cases the logic is embedded in the client-side component. These can be reverse engineered. JavaScript can be debugged line by line to identify embedded logic. This could include logic for cryptography algorithms, credential storage, privilege management and other security. This may lead to possible exploits.

**[SM-D01-R06]** To check for these kind of weaknesses, analyze the Document Object Model (DOM) and identify variables on a browser stack, and look for suspicious values and parameters that can be exploited in DOM.

### 8. Identity or profile extraction

A critical parameter in authenticated applications, the user's identity is maintained using session or other forms of tokens. Attackers can identify these token parameters in poorly designed and developed applications, opening up the potential for abuse and systemwide exploitation. The token may only be using a sequential number or guessable username.

**[SM-D01-R07]** To test for this, look for parameters that are controlling profiles; if it's possible to decipher, guess or reverse engineer tokens.

### 9. File or unauthorized URL access and business information extraction

Business applications contain critical information in their features, in the files that are exported and in the export functionality. Users can export their data in a selected file format (PDF, XLS or CSV) and download it. If this functionality is carelessly implemented, it can enable asset leakage.

**[SM-D01-R08]** To test for this, identify call functionalities based on parameter names like file, doc, and dir, which will point you to possible unauthorized file-access vulnerabilities, and then a good test is doing basic brute force or guesswork to fetch another user's files from a server.

### 10. Denial of service (DoS) with business logic

Denial-of-service vulnerabilities for business applications pose serious issues because if exploited, the application can be brought down for a length of time or at a critical juncture. Sometimes attackers can identify a loophole and try to exploit it during a DoS condition. There are no universal DoS attacks like TCP flooding on networking at the application layer, but in some cases, infinite loops implemented in the application layer can lead to a DoS condition.

**[SM-D01-R09]** Test applications against a threat model and provide defense at the application layer.

**JSON Web Token (JWT) libraries**

Critical vulnerabilities exist in several JSON Web Token (JWT) libraries – namely the JavaScript and PHP versions – that could let an attacker bypass the verification step. Attackers could exploit one of those vulnerabilities, which abuses an asymmetric signing algorithm, in some JWT libraries.  
  
JWT is a standard that produces tokens between two parties. For example, a server can produce an admin token, transferred in JSON, and signed by the server’s key. Clients can go on to use that token to verify the user is logged in as an admin. The issue revolves around a public key confusion between systems signed with the hash function HMAC and those signed with RSA. If a server is expecting a token signed with RSA, but actually receives a token signed with HMAC, it will think the public key is actually an HMAC key. HMAC secret keys are supposed to be kept private, while private keys are well, public. In this scenario if an attacker got access to a public key, through an API in some JWT libraries, they could use it as a token and the server would accept it.

**[SM-D01-R10]** To address the issue, verify that tokens with different signatures are set up to be rejected either via a whitelisting or blacklisting mechanism. The server should already know what algorithm it uses to sign tokens, and it’s not safe to allow attackers to provide this value.

A separate issue, since fixed in many JWT libraries, previously let attackers choose the way tokens are verified. This issue is rooted in the way that some libraries handled an algorithm known as “none.” Tokens signed with “none” could have be acknowledged as valid tokens with valid signatures. Attackers could modify tokens and sign them with “none” instead of HMAC-SHA256, or HS256. The tokens would then appear “signed.” Attackers then could have gone on to attach their own payload to gain arbitrary account access on some systems.

**[SM-D01-R11]** Fix the “none” issue by ensuring that token verification fails any tokens that use the “none” algorithm. Fix the attacker-controlled algorithm. JWTs can work across several languages, .NET, Node.js, Python, PHP, Java, Ruby, to name a few, and many remain vulnerable.

List of non-vulnerable libraries (partial): The issue is fixed in the Node.js library and users should upgrade to 4.2.2, the latest version. Jose Padilla, who maintains the Python build of the library, fixed the signature verification vulnerability in version 1.0.0 last month by adding support for an alg whitelist. The most recent version, 1.0.1, also includes the fix.

List of vulnerable libraries (partial): The PHP or JavaScript versions of the libraries remain vulnerable. Auth0 is instructing those who run those versions of JWT in particular to seek out another non-vulnerable library until the issues are fixed or verified. The Ruby version of the library is still vulnerable as well.

* A change to the specification, deprecating the header’s alg field, would ultimately benefit the community.