Client Input Filters

(a.k.a. Untrusted Client, Server-Side Validation, Sanity Checking)

Abstract

Client input filters protect the application from data tampering performed on untrusted clients. Developers tend to assume that the components executing on the client system will behave as they were originally programmed. This pattern protects against subverted clients that might cause the application to behave in an unexpected and insecure fashion.

Problem

Web applications are client-server applications, in which the client executes on untrusted hardware outside of the control of the Web application developer. Developers have a tendency to believe that the application will execute as programmed.

However, it is often trivial for attackers to tamper with Web clients, causing them to behave in an untrustworthy manner. For example, many sites depend on data validation performed by JavaScript functions executed on the client. It is easy to copy the page source, remove those checks, and execute the modified client code. Furthermore, the attacker can read the original code and learn what sanity checks the application applies to data.

Other sites rely on an intricate back-and-forth sequence of form submissions, in which data from earlier pages is stored in hidden fields on later pages. Again, it is trivial for the attacker to alter the contents of those fields before submitting the final page. In one famous case, a number of e-business sites used hidden fields to store catalog prices, so that a total selling price could be quickly computed on the client. Attackers were able to freely substitute prices reflecting extremely deep discounts.

A few sites even use client-side checks to enforce security measures, such as checking passwords, enforcing access control restrictions, or implementing account lockout after several failed password attempts. These measures cannot be trusted to execute properly, as attackers are free to ignore them.

A related problem is that many sites depend on anonymous users to submit data that cannot really be validated. For example, a site may ask users to provide their names and mailing addresses before a file can be downloaded or a physical catalog is mailed out. If a malicious user provides bogus data, the site might attempt to process the data. This can provide an avenue for resource consumption attacks and, in the case of the physical catalog mailing, can be quite expensive. Even if detected, a site flooded with spurious requests may cause valid transactions to be discarded along with the invalid ones.

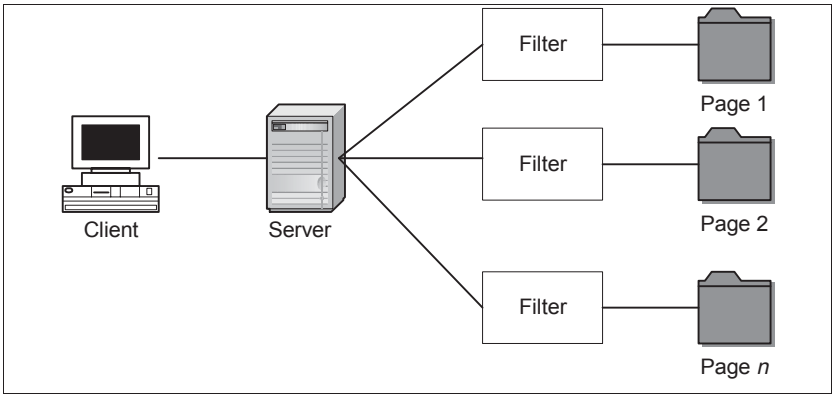
Solution

All data provided by the client should be treated as suspect and filtered at the server:

- Calculated fields provided by the client should be ignored and recomputed at the server when the data is processed.
- Data validity checks performed on the client should be repeated on the server before the data is processed.
- Sensitive data that must be stored on the client should be kept in an encrypted, tamper-proof form. (See the *Client Data Storage* pattern.)
- In addition to field-level validity checks, the server should look for specific signs of bogus data submission and discard requests that are obviously questionable.
- Suspiciously long URLs and header fields should be dropped (and possibly logged).
- Text data that is submitted by the user should be filtered to eliminate scripting tags and other questionable content.

Sanity checks can be integrated into standard page objects, or they can be implemented in separate objects that use chaining (or servlet filtering) to intercept and modify the requests as they are dispatched to the page objects. The latter approach is less efficient, but is easier to get right. In either case, the client filters should always be invoked before any processing of the client-supplied data occurs.

Client filters should be able to modify requests before delivering them to the intended object. If the data cannot easily be fixed, the client filter should reject or simply drop the request. All filtering events should be reported to the central logging mechanism – although many will be benign, they might indicate a pattern of attempted misuse. If a filter detects an obvious attempt to sidestep the security of the system, the request should be blocked and the event reported.



Issues

If an authenticated user account is exhibiting signs of tampering with the client or client-resident data, one possibility is that the account has been compromised. The legitimate user should be contacted out-of-band, or the account should be disabled and the legitimate user will eventually contact customer service.

Many Web sites use hidden fields on the browser, or client-resident cookies, to maintain client state. This can be more efficient than storing that state on the server. However, it is trivial for users to inspect the contents of these fields. In particular, plaintext passwords should never be stored in cookies, URLs, or browser pages.

When testing a Web site, there are a number of techniques that can be used to see all the content that is usually invisible. Use "view source" to examine hidden fields, use "prompt for cookies" to inspect the contents of cookies, and install a packet sniffer to examine the contents of header fields. If using SSL, install a proxy so that the packet sniffer will be able to see decrypted packets between the browser and the proxy. All of these techniques help you understand how easy it is for attackers to look under the surface of an application.

Log all suspicious client behavior, including the originating network address. These logs may be critical for later prosecution if the client turns out to be an attacker. Furthermore, these logs should be integrated into the Intrusion Detection System in order to inform the system administrators that the site may be under attack.

Duplicating Client Side Checks

Web sites often use client-side JavaScript to validate form submissions (either that individual fields comply with formatting restrictions, or that specific fields have not been left empty). It is trivial to bypass these checks. Some user firewalls and browsers even filter out JavaScript, resulting in forms that work but are never validated. For these reasons, it is necessary to re-validate all user-supplied data on the server. The client-side validation is a useful efficiency: it provides greater response times and reduces the burden on the server. But it should never be depended on.

Sensitive data calculated on the browser should be validated on the server. Some e-commerce sites store unit pricing information on catalog Web pages, and then depend on the client browser to calculate total prices. This allows the Web page to be more responsive to user changes then one that must go back to the Web server for these calculations. However, it is critical that the results of that calculation be validated on the server.

When replicating client-side checks on the server, it is crucial that the computations be consistent. Continuing the previous example, if the server recalculates the total price and computes a value that is higher than that calculated by the client, users will be very, very upset. If there is any chance that the calculations could become inconsistent, it is better to take the performance hit and simply calculate everything on the server.

Sanity Checks

For any transaction request that can be initiated by an unauthenticated user, the Web application should invoke a sanity-checking module before processing the request. A sanity-checking module will conduct checks such as: do any of the fields not comply with field-level checks that should have been performed at the client? Is the request from an IP address that has made similar requests recently? Is the request largely similar to another recent request? Do any of the fields contain random garbage (e.g., "asdf" or other obvious sequences), or do multiple fields appear suspect? Do specific field contents raise red flags (e.g., e-mail address of

"president@whitehouse.gov") Using this approach, it should be possible to filter out a large proportion of obviously bogus submissions. Other questionable submissions can be flagged for later observation by a human operator.

Filtering Possible Attacks

There are products that allow data submitted to the Web server to be filtered according to predefined rules. For example, if any legitimate URL is no more than 256 characters, it would be appropriate to truncate any longer request (which may contain a buffer overflow attack). If binary files are never uploaded to the server, non-printable ASCII characters could be stripped from all requests (again, thwarting potential buffer overflows).

Many sites allow users to upload text to be posted on the site (e.g., product reviews, general message forums, auction postings, news stories). Users who know HTML enjoy adding tags to their text in order to add formatting or draw the reader's attention. But any site that echoes user-supplied HTML could be misused to attack other sites. Filter out anything but the most basic HTML tags (font formatting). If links to other sites are provided by users, be careful to filter out any scripting tags or escape sequences. There are libraries available to filter out *cross-site scripting attacks* – use one.

Possible Attacks

There are a number of possible attacks that could be perpetrated against this pattern:

- Cross-site scripting attacks – this attack occurs when a Web server echoes user-provided JavaScript containing links to a hostile site. For example, many sites include user comments and reviews. If the site does not filter input, a client can be tricked into divulging sensitive cookie data to the cross-linked site.

- Resource consumption attack against the server – if the server is required to keep state for each client, this can lead to depletion of resources on the server if every new client consumes some finite resource on the server.

- Automated submissions – it is straightforward to develop an automated tool for making large numbers of form submissions to a Web site. There are even books explaining how to build "votebots" that can skew on-line polls [6]. The most dangerous of these are password-guessing attacks and enrollment secret-guessing attacks.

Examples

Operating systems do not trust the applications running on them. When applications attempt to invoke operating system services, the operating system checks that the parameters are all valid before executing the command. Operating systems also store critical data within protected memory, providing only opaque handles to the application [5].

As an efficiency, early versions of Windows did not perform parameter validation, and the results were a significant decrease in stability.

At the network level, modern e-commerce sites have generally learned to employ these techniques in order to avoid repeating the earlier mistakes of others. Amazon.com, for example, maintains shopping cart data and all pricing information on the server. The client contains only a session identifier. Session cookies are apparently used to track user browsing habits, but not anything critical to the shopping transaction. Permanent cookies are used to identify the user upon repeat visits, but do not authenticate the user. (One-click shopping, if enabled, is an exception to this.)

This pattern is commonly used in almost all Web applications to verify e-mail addresses; anything entered in an e-mail address field that is not valid will result in a warning.

On at least one Microsoft Web site that attempts to gather customer information, any attempt to enter an e-mail address ending in "microsoft.com", "msn.com" or "msnbc.com" will result in the submission being rejected. This appears to be an instance of sanity checking.

Trade-Offs

Accountability	No effect.
Availability	If overly sensitive, this pattern can have an adverse effect on availability, preventing legitimate users from using the site.
Confidentiality	No effect.
Integrity	This pattern greatly enhances the integrity of the data processed by a Web site.
Manageability	The management burden could be increased if overly sensitive sanity checks result in a high number of false reports of attacks that must be investigated.
Usability	No effect.
Performance	This pattern will incur a small performance penalty, since it requires some time to perform checks. If data is stored in encrypted form on the client, encrypting and decrypting the data will also exact a performance hit.
Cost	This pattern has fixed implementation costs. However, if overly sensitive it could greatly increase the customer service burden on the site.

Related Patterns

- *Network Address Blacklist* – a related pattern the responds to suspicious client behavior; if

any client-side validation fields have been obviously disabled, this should be input to the network address blacklist.

References

[1] Dominy, R. "Focus on JavaScript: Email Field Validation". http://javascript.about.com/library/scripts/blemailvalidate.htm, 2002.

[2] INT Media Group. "Email Address Validation". http://javascript.internet.com/forms/check-email.html, 2002.

[3] Open Web Application Security Project (OWASP). "Input Filters". http://www.owasp.org/filters/index.shtml, May 2002.

[4] Peterson, S. and D. Fisher. "The next security threat: Web applications". http://zdnet.com.com/2100-11-503341.html?legacy=zdnn, January 2001.

[5] Silberschatz, A., J. Peterson, and P. Galvin. *Operating System Concepts Third Edition*. Addison-Wesley, 1991.

[6] van der Linden, P. *Just Java 2 – Fourth Edition*. Prentice Hall, 1999.