SQL Smuggling

Or,
The Attack That Wasn't There

Comsec Consulting Research
By Avi Douglen
Senior Consultant

Reviewed by Shay Zalalichin
AppSec Director

November, 2007

Introduction

SQL Injection is a common, well-understood application-level attack that misuses the trust relationship between the application and the database server. This relationship is exploited in order to attack the database directly through the application, or to use the database server in order to attack other systems in the organization. Several applicative mechanisms exist for protection against SQL Injection attacks, including input validation and use of Stored Procedures.

This paper will present a new class of attack, called SQL Smuggling. SQL Smuggling is a sub-class of SQL Injection attacks that rely on differences between contextual interpretation performed by the application platform and the database server.

While numerous instances of SQL Smuggling are commonly known, it has yet to be examined as a discrete class of attacks, with a common root cause. The root cause in fact has not yet been thoroughly investigated; this research is a result of a new smuggling technique, presented in this paper. It is fair to assume that further study of this commonality will likely lead to additional findings in this area.

SQL Smuggling attacks can effectively bypass standard protective mechanisms and succeed in injecting malicious SQL to the database, in spite of these protective mechanisms. This paper explores several situations wherein these protective mechanisms are not as effective as assumed, and thus may be bypassed by malicious attackers. This in effect allows an attacker to succeed in "smuggling" his SQL Injection attack through the applicative protections, and attack the database in spite of those protections.

Variations between platforms include well-known differences, such as product-specific syntax. Likewise, previous research has been done, extensively examining techniques for evading signature-based detection. However, those papers describe specific techniques, without examining the common root cause mentioned above. In addition, we will describe a new addition to this class of attack, based on differences in the way certain database servers handle Unicode homoglyphs [1].

The basic premise asserted by this paper is that more attention needs to be paid to the inherent logic employed by the database server. This is contrary to common attitudes, where the database is assumed to be a static datastore that does whatever it is told to do. In fact, database servers are complex beasts, and often implement their own layer of interpretations. As such, validation checks need to be as restrictive as possible, taking the context of the specific datasource into consideration.

Moreover, additional research in this area is required, further examining the interaction between the application and the database platform, explicitly exploring the conversions that take place during the course of communication. Database vendors should also consider this interaction, and take appropriate steps to mitigate any implicit effects.

Background

SQL Injection Prevention

It is commonly accepted that strong input validation can prevent SQL Injection attacks. Of course, additional mitigations are often recommended as best practices, such as only accessing Stored Procedures, using appropriate Command / Parameter objects (in programming languages that support them), Least Privilege, etc.

While these are all good ideas for defense in depth, the chief safeguard against malicious input is always validating input. Specifically, OWASP defines [2] proper validation as including the following steps:

1. Escape apostrophes (and other meta-characters), e.g. with two apostrophes;
2. Ensure numeric fields really look like numbers;
3. Check if the inputs are within your expectation (e.g. 0 < age < 120, login Id without space, etc.);
4. The apostrophe-escaped input should NOT be further en/decoded by any other coding scheme.
5. Additionally, it is recommended best practice to block input that "looks" like SQL, such as "UNION SELECT", "INSERT", etc. This can add an additional layer of "Defense In Depth" for some scenarios.

Traditional Smuggling Attacks

Smuggling attacks are based on sneaking data past a point where prohibited without detection (e.g. across a network, or into a Web server), by "hiding" the data, often in plain sight. For instance, this includes causing the data to look like permitted data. Another method would be to pass irrelevant data past the prohibited point, in a format that can be reconstructed into the forbidden data once it has safely passed the detection point.

A lot of research has been done in this area, specifically pertaining to detection evasion [3]. One specific class of smuggling attack was discovered several years ago, HTTP Request Smuggling (HRS) [4]. This attack relies on differences between the ways various entities implement the HTTP RFC, and resulting discrepancies in interpreting malformed HTTP requests [5].

It is important to note that HTTP Request Smuggling is based on the fact that the second, malicious, request *does not exist* as an independent request entity, in the context of the HTTP intermediary (e.g. Web proxy, cache, etc.). This is in contrast to typical smuggling techniques, wherein the malicious data exists, but it is simply in a different textual format so as to be unrecognizable.

Introducing SQL Smuggling

Consequently, our research, together with discoveries in the field, has shown the necessity to define a new sub-class of SQL Injection attacks: SQL Smuggling. We define SQL Smuggling as SQL Injection attacks that can evade detection, as a result of different interpretations of the malicious data. Specifically, protective mechanisms – including data validation by the application, Web Application Firewalls (WAF), IDS, etc. – will not recognize or block the malicious input; however, the database server will in fact accept the submitted input as valid commands.

As described above, smuggling attacks can be based on disguising the malicious data to appear as permitted data to the validation mechanisms, but as the actual malicious payload to the server. Additionally, the smuggling attack may mutate the malicious payload in such a way that it would not exist in any meaningful form, within the context of the validation; and yet the server would transform the mutated data back to the malicious payload. Similarly, SQL Smuggling includes attacks based on disguising the payload from the validation mechanisms, and also attacks based on mutating the payload to a non-malicious form, that will be actively transformed by the database into the malicious payload.

The reason that it is possible to either disguise or mutate the malicious payload into a form that is not recognized by the validation mechanisms, but can still be properly understood or translated by the database server, is differences between the parsing engine of the validation mechanism and the database engine itself.

The causes of these differences can include various factors, only few of which could be expected by the application developer. Many such differences are a result of proprietary extensions not defined by the RFC, implemented differently by each specific database vendor. Certain factors may be implicit and obscure, and as such programmers may often not be aware of these factors, and thus will not apply appropriate mitigations. Other issues may even not be widely known, occurring internally to the database engine. As such, the validation mechanisms, whether they be implemented as applicative checks, WAFs, or IDS, interpret the data in a different manner than the database server itself.

Regardless of the specific source of discrepancy between the platforms, SQL Smuggling includes SQL Injection attacks that *do not exist* in the context in which the validation checks are performed.

This, then, is the chief distinction between SQL Smuggling and standard SQL Injection, and the rationale for a separate designation: standard protections against SQL Injection, such as data validation as described above, will not thwart SQL Smuggling. Typically data validation is based on *textual interpretation*, and validation thereof; however, database engines do more than simple textual parsing, and employ additional internal logic.

For this reason, it will never be possible to block a closed list of SQL Smuggling vectors, since to do so it would be necessary to fully implement all internal logic for each database platform. Often, much of this logic is not even publicized. This goes far beyond evasion of textual signatures, since in this case the string data does not even contain the textual payload, in any form whatsoever. This point should be emphasized: once the malicious data is mutated, searching for strings of any sort will fail, since the strings do not exist at the time of validation, but are "created" by the database server itself. Simply put, you cannot block data that is not there.

Simple SQL Smuggling

There are several forms of well-known attack vectors, which are commonly known by other terms. According to the above definition, these too can be seen as SQL Smuggling, though these were not denoted as such until now. Following are some examples of well-known SQL Injection vectors, which are in essence SQL Smuggling.

1. Platform-Specific Syntax

   Most database servers support extensions to standard SQL syntax, in addition to proprietary built-in functions and commands. Specifically, MySQL supports escaping meta-characters through the use of the backslash character ("\"). For instance, on MySQL a quote can be escaped in at least two ways, either double-quoting (`''`), or backslash and quote (`\'`). These are both valid escape sequences, and can be safely embedded within strings.

   If application code validates user input by simply escaping quotes, without recognizing the specific context of MySQL, the validation mechanism may allow actual quotes to pass unescaped. This may occur if the programmer is not familiar with the specifics of MySQL, or if the application architecture supports configurative definition of the database server and database type at runtime. External devices, such as a WAF, support only a limited number of dialects, and thus may not recognize a given variation – the classic drawback of blacklisting forbidden values, i.e. it is always possible that there will be another dangerous value that is supported by an uncommon database product.

   For instance, if the validation mechanism is based (amongst other factors) on doubling any quotes, the "`\'`" sequence will be translated to "`\''`". This will be understood by MySQL as: "`\'`" (escaped quote) followed by `'` (unescaped quote). The result is, of course, successful SQL Injection.

   This is the most trivial of SQL Smuggling attacks, wherein application code is not aware of the context within which the input data will be interpreted. In actuality, there may be additional instances of slight differences in syntax between database vendors, such as differing commands or functions between vendors' proprietary extensions to ANSI SQL (e.g. MSSQL's T/SQL, Oracle's PL/SQL, MySQL, PostgreSQL, etc.). The proprietary built-in functions may also be exploited unless specifically searched for and blocked by the validation mechanism, e.g. MSSQL's `OPENROWSET` command. Again, if the application's validation code is sensitive to a single variant, but applied to a different one, the validation will obviously not succeed.

   It should be noted that a proper WAF product would recognize, and appropriately block, the chief variants, such as those for the most popular platforms including the above mentioned MySQL issue, in addition to MS-SQL, Oracle, DB2, and perhaps other common products. However, while this may provide some protection in most cases, it is still improbable to expect the WAF to recognize and be familiar with all possible variants, including those supported by less popular database products that are not as well-known. Moreover, it can be expected that the WAF might not cover every possible aspect of a given SQL dialect, not to mention that there may be conflicts between interpretations depending on choice of dialect.

2. Signatures Evasion

   Many network protection devices, such as IDS/IPS and WAFs, base much of their protection on recognizing attack patterns, such as input data containing SQL-like strings. Numerous methods have been published [6] to evade detection and prevention including:

   - Innovative use of whitespace (space, tab, newline…);
   - Inline comments (using `/*…*/`);

- Different encodings (URL encoding, Hex encoding, etc.);
- Dynamic concatenation and execution of string values (e.g. `CHAR()` or "`EXEC ('INS' + 'ERT INTO…')`").

Using any of these techniques, it is possible to create valid SQL commands that are understood and accepted by the database's parsing engine, and yet are not recognized by the validation checks. Note that it is not relevant if these checks are performed by the application code, application firewall, IDS/IPS, or any other validation entity – in any case, they attempt to validate the string inputs based on an explicit blacklist of malicious commands or patterns; this blacklist is finite and cannot specify every possible attack.

Once again, the attacker can succeed in penetrating system defenses to perform SQL Injection, by smuggling the payload in a form that is not explicitly blocked by the validation checks. Note that there are vast possible combinations and permutations of malicious data that would evade the validation checks. These are also partially dependant on differences between the database platforms and versions.

Unicode Smuggling

Theory

There are numerous Unicode characters that are "similar" to other characters, such as those in the base ASCII character set. These are known as homoglyphs.

Our research discovered a surprising finding: some database servers support automatic translation between supported codepages. E.g. a Unicode value may be automatically converted to a different character in the local character set, according to a "best fit" heuristic, even though these characters are not equivalent and are computationally unequal.

When string data containing these characters are forced from Unicode into another character set, the database server may perform the "best fit" matching, translating some of these characters to whatever character is deemed visually equivalent in the target codepage. For instance, the Unicode Ā character (U+0100) may be translated to the regular ASCII A (U+0041). As surprising as this is, this automatic translation occurs for numerous other characters, as many homoglyphs exist.

Since this is performed by the database, any mechanism that attempts to filter or sanitize values before passing them to the DB may fail to recognize certain characters, which will be "created" as a result of the transformation by the database server. Again, this translation occurs after all validation checks.

And this is where the interesting (or scary) part starts – the Unicode homoglyph translation is not limited to base alphabet characters... Specifically, the Unicode character U+02BC (modifier letter apostrophe) can be translated by the database server to a simple quote – ' (U+0027). There are, of course, many other similar examples.

The result of this translation is quite clear – certain strings can evade applicative filters, such as those intended to validate input on the application server and prevent SQL Injection. Though the string is seemingly innocent when validated by the application, it is potentially dangerous when these homoglyphs are translated by the database. In particular, since a quote can bypass application filters and yet be recognized by the database, malicious input may "break out" of quoted parameters, which obviously can result in SQL Injection attacks.

Our research in this area shows that this is the most extreme example of SQL Smuggling, since the *actual character itself* does not even exist at the time of validation. In reality, the meta-character is created by the database server itself, as a result of the homoglyphic transformation. As such, no form of applicative checks would detect this character.

It should be noted that, although this specific vector can likewise be blocked (i.e. disallow the U+02BC character), this is solely a representative example; it is not feasible to explicitly block all such database behavior, both expected and unexpected, for each database platform and version, undocumented features and version-specific bugs included.

Technical Details

As an example, note the following trivial Stored Procedure:

```
create procedure GetData ( @param varchar(20) ) as
begin
declare @s varchar(200)
select @s = 'select * from dataTable where name = ''' + @param + ''''
exec (@s)
end
```

This SP may be called from a Web page, which executes validation code before passing the input to the SP. At a minimum, this validation code either verifies that the input does not contain a quote, or sanitizes it to double any existing quote. For instance, the validation code may be using `string.Contains()`, `string.Replace()`, Regular expressions, etc. It is also possible that this Web page is behind a finely-tuned Web Application Firewall that validates all input and verifies that no quotes are included.

A malicious user or attacker can submit malicious code containing a modifier letter apostrophe (U+02BC, URL encoded to %CA%BC). This will easily pass applicative validation code and WAF filters, since these search for an actual quote (U+0027) which does not exist in the input at this time. Obviously, IDS/IPS systems would also not detect anything amiss.

The validation mechanisms may even search for various encodings of a quote, such as URL Encoding, UTF-8 encoding, Hex encoding, double encoding, and more – however, U+02BC is none of these, and is in fact a completely different character value.

Even if the application calls the SP properly, using parameter objects etc., the input will be passed to the SP in a `varchar` variable, which typically does not support Unicode values; an affected database server will promptly translate this character to a standard quote, breaking the `WHERE` clause and causing the dynamic execution to execute whatever arbitrary code was included in the malicious parameter.

It could be noted that the SP in this example does utilize dynamic SQL execution, which should be avoided (according to accepted Best Practice). However, there are certain scenarios wherein server-side dynamic SQL is absolutely required, e.g. legacy code, third party products that include code on the database, and more. Moreover, our experience in the field shows that many real systems support these complex scenarios, and do in fact require dynamic SQL in SPs. Aside from that, though popular, it is intended only to serve as yet another representative example. That said, the typical recommendation for allowing dynamic SQL within SPs (when they are absolutely necessary), is of course to perform complete input validation before calling the SP.

This research voids that recommendation, and puts the more complex scenarios at risk. Till today, input validation was considered by domain experts to be an adequate technique, sufficient to protect against SQL Injection attacks; however, this is no longer the case!

Applicability

SQL Smuggling can apply to many applications that rely on input validation, either by applicative code or a WAF. However, not every vector is applicable in all cases: for example, the MySQL backslash escaping is obviously only relevant to systems based on the MySQL database, and only if the programmer did not take appropriate steps relevant to the specific database. Additionally, many signature evasion techniques may not apply to every system.

However, by definition SQL Smuggling can evade detection, and as a general class would apply to virtually any system relying solely on blacklist validation mechanisms. It is likely that

additional forms of Smuggling will be discovered, further broadening the scope. Those applications that are built on a secure design, and implement a strict white-list validation mechanism, should be immune to this class of attack.

Regarding Unicode-based SQL Smuggling, this flaw can clearly cause severe damage in certain situations, though research to date shows that it is currently limited to specific scenarios, and would probably not be vulnerable if the application had been built according to best practices in the first place.

However, imperfect applications such as these are numerous, and we have observed numerous real-world examples in the field. (If only everyone built everything the correct way…). Furthermore, it is possible that there are complex situations where a design such as this may be required from the application.

Another important factor to consider is the database platform. Automatic, "best-fit" homoglyphic transformations are a feature intended to ease user's pain in translating between "foreign" languages and the local character set. It seems that this homoglyphic transformation is not supported by all databases. Specifically, Microsoft's SQL Server 2005 does support homoglyph transformation. Older versions of MySQL supported a version of the Connect/J connectivity library that apparently performed this transformation. We are still researching other databases.

Recommendations

The only means of preventing SQL Smuggling, in situations where the application would be otherwise vulnerable, is performing context-based validation. That is, validation checks must take the characteristics of the database platform into account, including the brand of database server, defined character sets, and more. (Note that this negates loosely-coupled architectures, and as such would be even more difficult to implement properly.)

Moreover, validation should be based on a "White-list" of allowed characters. While this has always been a recommended best practice, many implementations of "Black-list" seemed sufficient, and impervious to SQL Injection attempts. This research proves that besides being inefficient and usually incomplete, it can never suffice. Any validation that stops short of whitelisting the allowed characters (e.g. [a..zA..Z0..9]), will not be complete.

Avoid dynamically concatenating SQL queries for execution by the database. This applies both to application code and stored procedures. Wherever user input is used directly in a query, there exists a risk of that input bypassing the applied validations and disrupting the intended command.

Additionally, never forcefully translate strings from one character set to another. Results may be unpredictable, and are rarely beneficial. Wherever possible, it is best to remain in a single character set. If necessary to support more than a single language, the database should be built using Unicode.

This research presents only the "tip of the iceberg"; we therefore recommend that other security researchers take part and expand the research done in this area. The interaction between the application and the database platform should be further examined, and the myriad conversions that take place during the course of communications must be explicitly explored.

As for vendors of database software, connectivity libraries, and any of the other myriad components that play a part in accessing data, we recommend redesigning those products to minimize implicit database transformations and other platform-specific logic; specifically, remove support for automatic homoglyph transformation. These products should also be examined for other behaviors that allow malicious payload to be generated by the server.

Conclusion

Our research discovered a new vector of attack, and as such defined a new sub-class of attacks. **This paper proves that an application may be vulnerable to SQL Injection attacks, even though it does proper input validation before calling a stored procedure**, in contrast to conventional wisdom that input validation is sufficient to protect an application against SQL Injection. This is of course dependant on the application structure and format of queries; as such, if an application does follow all relevant best practices (such as not using Dynamic SQL, etc.), it is mitigated to some extent.

Contrarily, attention should be given to the possibility of evasion of Application Firewalls, Intrusion Detection/Prevention systems, etc. In effect, an attacker may exploit vulnerable applications to "smuggle" SQL commands through all the defensive perimeters. As such, more attention needs to be paid to database server logic, along with any logic performed by components used to access the database.

Thus, the basic premise is well founded, as more attention needs to be paid to the inherent logic employed by the database server. This is contrary to common attitudes, where the database is assumed to be a static datastore that does whatever it is told to do. In fact, database servers are complex beasts, and often implement their own layer of interpretations. As such, validation checks need to be as restrictive as possible, taking the context of the specific datasource into consideration.

Moreover, more research should be done in this area, further examining the interaction between the application and the database platform, explicitly exploring the conversions, transformations and other logic that take place during the course of communication.

References

1. Homoglyph, Wikipedia
   http://en.wikipedia.org/wiki/Homoglyph#Unicode_homoglyphs
2. "Advanced Topics on SQL Injection Protection", Sam NG, OWASP Feb 2006
   http://www.owasp.org/images/7/7d/Advanced_Topics_on_SQL_Injection_Protection.ppt
3. "A Look at Whisker's Anti-IDS Tactics", Rain Forest Puppy, December 1999
   http://www.ussrback.com/docs/papers/IDS/whiskerids.html
4. "HTTP Request Smuggling",
   Chaim Linhart, Amit Klein, Ronen Heled and Steve Orrin, June 2005
   http://www.modsecurity.org/archive/amit/HTTP-Request-Smuggling.pdf
5. "Meanwhile, on the other side of the web server", Amit Klein, June 2005
   http://www.modsecurity.org/archive/amit/meanwhile_on_the_other_side_of_the_web_server.txt
6. "SQL Injection Signatures Evasion", Ofer Maor and Amichai Shulman, April 2004
   http://www.imperva.com/application_defense_center/white_papers/sql_injection_signatures_evasion.html
7. "The Unexpected SQL Injection", Alexander Andonov, January 2007
   http://www.webappsec.org/projects/articles/091007.shtml

About Comsec

Comsec Consulting (TASE:CMSC) is a leading security consulting company, helping enterprises design and incorporate security into their information technology infrastructure and systems. For over two decades, Comsec has delivered cutting edge, end-to-end information security services to customers from all market sectors across the globe, including major high-tech firms, telecom operators, major banks and financial institutions, government bodies and leading industrial corporations. Comsec Consulting provides risk assessments, security planning and design, develops security policies, procedures and SDLC, performs threat modeling, code reviews and penetration testing, in addition to security education & training. Comsec Consulting provides these services internationally through offices in London, Rotterdam, Istanbul, Warsaw and Tel Aviv. Please visit us at: www.ComsecGlobal.com.